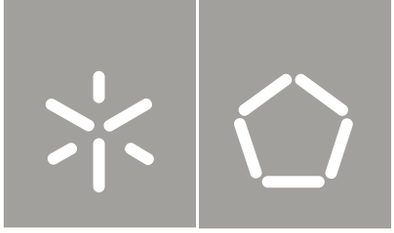
Universidade do Minho
Escola de Engenharia

Luís Miguel Nascimento Duarte

Simulação de Comunicações Oportunistas em Ambientes Urbanos Utilizando Bluetooth

outubro de 2013

Simulação de Comunicações Oportunistas em Ambientes Urbanos Utilizando Bluetooth
Luís Miguel Nascimento Duarte
UMinho | 2013



Universidade do Minho
Escola de Engenharia

Luís Miguel Nascimento Duarte

Simulação de Comunicações Oportunistas em Ambientes Urbanos Utilizando Bluetooth

Tese de Mestrado
Ciclo de Estudos Integrados Conducentes ao
Grau de Mestre em Engenharia de Comunicações

Trabalho efetuado sob a orientação do
Professor Doutor Adriano Moreira
Professora Doutora Maria João Nicolau

outubro de 2013

DECLARAÇÃO

Nome: Luís Miguel Nascimento Duarte

Correio electrónico: lmnd88@gmail.com

Tlm.: 960121229

Número do Bilhete de Identidade: 13353581

Título da dissertação:

Simulação de Comunicações Oportunistas em Ambientes Urbanos Utilizando Bluetooth

Ano de conclusão: 2013

Orientadores:

Professor Doutor Adriano Moreira

Professora Doutora Maria João Nicolau

Designação do Mestrado:

Ciclo de Estudos Integrados Conducentes ao Grau de Mestre em Engenharia de Comunicações

Escola de Engenharia

Departamento de Sistemas de Informação

Guimarães, / /

Assinatura: _____

"Knowledge speaks, but wisdom listens."

– Jimi Hendrix

AGRADECIMENTOS

Sendo este o ponto final, ou quem sabe, apenas um ponto e vírgula no meu percurso académico, é um momento importante no meu crescimento pessoal e profissional, o qual não seria possível sem o acompanhamento e ajuda de muitos que contribuíram de forma direta e indireta para que todo este caminho fosse percorrido, e alcançada a meta.

Tenho então que desde mais agradecer aos meus orientadores Professor Doutor Adriano Moreira, e à Professora Doutora Maria João Nicolau, que se mostraram incansáveis no apoio prestado, sempre disponíveis a ajudar e a dar indicações no rumo a tomar, no qual estou extremamente grato.

Ao meu pai António Luís e à minha mãe Maria Helena, que sempre me apoiaram quer fosse na vida académica ou pessoal, e que sempre tudo fizeram para que este momento fosse possível. Para eles um muito obrigado por tudo.

Ao meu avô Nascimento que sempre esteve a meu lado, preocupado e interessado, procurando sempre ajudar-me no que podia e no que não podia, qualquer que fosse a situação. Para ele um abraço do tamanho do mundo.

Gostaria de agradecer também ao Nuno Correia, que partilhou comigo o desenvolvimento deste projeto, e que se mostrou sempre disponível a ajudar. Agradeço assim a sua entreajuda e companheirismo.

Não poderia de deixar agradecer também, a todos os companheiros desta vida académica, que fizeram com que esta se tornasse menos monótona, nomeadamente ao Hugo Leite, João Brito, José Teixeira, Laurent Miranda, Manuel Rocha, Pedro Machado, Ricardo Maciel, Rui Rodrigues, Tiago Pimenta, e em especial ao Cristiano Pendão e Luís Gonçalves, insubstituíveis companheiros de estudo e grupo ao longo destes anos.

RESUMO

Assistimos nos dias de hoje nos meios rurais mas maioritariamente nos meios urbanos, um enorme crescimento do uso de dispositivos móveis. Pedestres ou mesmo veículos, passaram a integrar ou transportar dispositivos com sistemas de comunicações como telemóveis, tablets, computadores portáteis ou até relógios inteligentes que trazem a possibilidade de comunicarem entre si. Dessa forma torna-se importante que existam aplicações capazes de simular o movimento e a interação entre estes dispositivos.

Já existem atualmente diversos simuladores de mobilidade e comunicações que procuram simular ambientes urbanos, contudo não existem ainda aplicações capazes de fazer simulações em grande escala ou simular específicos protocolos de comunicação. Foi com o intuito de responder a estas lacunas, que foi pensado e nasceu este simulador. É assim objetivo que este para além de fazer simulações de mobilidade para diferentes tipos de atores, consiga também fazer a simulação da comunicação entre estes.

Esta dissertação descreve o desenvolvimento de um simulador de comunicações em ambientes urbanos que permita assim simular as comunicações entre os diversos atores intervenientes numa simulação. É pretendido que estas consigam trocar informação entre si, usando para isso mecanismos de simulação tendo em conta pormenores da vida real, tal como o alcance rádio da tecnologia usada, ou a possibilidade de envio de mensagens Broadcast. Foi então simulada a tecnologia de comunicação Bluetooth, e efetuados testes de desempenho e funcionalidade que vieram assim concluir que os objetivos pretendidos desta dissertação foram alcançados.

ABSTRACT

Nowadays in the rural but mostly in the urban areas, the use of mobile devices it's growing hugely. Pedestrians or vehicles have been incorporating and carrying devices with communication systems such as mobile phones, tablets, laptops or even smart watches that provide the ability to communicate between each others. Therefore it's important that exists applications capable to simulate the movement and the communications between these devices.

There are some mobility and communications simulators that simulate urban environments, however the applications capable of doing large-scale simulations or specific communication protocol simulations are still absent. This simulator was born in order to fill these gaps. It's aim of this project that beyond capability of mobility simulations for different types of actors, this simulator became also capable to simulate the communications between them.

This master thesis then describes the development of urban environment communications simulator, capable of simulate the communications between the different actors in a simulation. It's intended so, that the actors be able to exchange information between each other, using for that simulation mechanisms taking into account details from real life, as the radio range from the used technology, or the ability to send broadcast messages. Was then simulated the Bluetooth communication technology and performed performance and functionality tests, which thus conclude that the intended goals from this dissertation were achieved.

CONTEÚDO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE LISTAGENS

LISTA DE ACRÓNIMOS

ACL Asynchronous Connection-Less

API Application Programming Interface

BartUM Bartolomeu Urban Mobility Simulator

CAD Computer Aided Design

CBR Constant Bitrate

CRC Cyclic Redundancy Check

CSV Comma-separated values

CTS Clear to Send

DAC Device Access Code

DCD Data Carrier Detect

DLCI Data Link Connection Identifier

DPSK Differential Phase-Shift Keying

DQPSK Differential Quadrature Phase-Shift Keying

DSR Data Set Ready

DTN Delay-Tolerant Networking

DTR Data Terminal Ready

DUN Dial Up Networking

EDGE Enhanced Data rates for GSM Evolution

EDR Enhanced Data Rate

FEC Forward Error Correction

FHS Frequency Hopping Synchronization

FHSS Frequency-Hopping Spread Spectrum

GC Global Coordinator

GFSK Gaussian Frequency-Shift Keying

GIS Geographic Information Systems

GMSF Generic Mobilitie Simulation Framework

GSM Global System for Mobile Communications

GUI Graphical User Interface

HCI Host Controller Interface

IAC Inquiry Acccess Code

IEEE Institute of Electrical and Electronics Engineers

IP Internet Protocol

ISM Industrial, Scientific and Medical

L2CAP Logical Link Control and Adaptation Protocol

LAN Local Area Network

LC Local Cordinator

NS-2 The Network Simulator 2

OD Origin-Destination

OSI Open Systems Interconnection

OSM Open Street Map

PAN Personal Area Network

PCMCIA Personal Computer Memory Card International Association

PSM Protocol and Service Multiplexor

QoS Quality of Service

RAM Random Access Memory

RDMA Remote Direct Memory Access

RFCOMM Radio Frequency Communication

RTS Request to Send

SCO Synchronous Connection Oriented Link

SDP Service Discovery Protocol

SIG Special Interest Group

SUMO Simulator of Urban MObilitie

TCP Transmission Control Protocol

TDM Time-Division Multiplexing

UI User Interface

VOIP Voice Over IP

XAVE Wireless Access in Vehicular Environments

XML eXtensible Markup Language

WKT Well Known-Text

1 INTRODUÇÃO

1.1 ENQUADRAMENTO

Nos dias de hoje a maior parte dos dispositivos móveis pessoais como telemóveis, smartphones, tablets, netbooks ou até veículos, possuem não só capacidades de se ligarem à internet, mas também a capacidade de estabelecerem comunicação com outros dispositivos próximos. Com esta evolução o estudo do desempenho destes sistemas de telecomunicações móveis tal como a própria dinâmica dos espaços, torna-se cada vez mais importante, notoriamente em áreas urbanas. Começaram então a ser estudados sistemas simuladores, com o intuito de se obter resultados face à dinâmica e interação entre dispositivos móveis, o mais próximos da realidade possível. Os sistemas simuladores atualmente existentes, continuam muito genéricos, não permitindo a avaliação de um cenário urbano em particular.

Em 2010 começou a ser desenvolvido o Bartolomeu Urban Mobility Simulator (BartUM), um simulador de espaços urbanos em que as características de um espaço real, como os passeios, ruas, sentidos de transito, semáforos, paragens de autocarro e edifícios entre outros, são condicionantes do movimento das entidades móveis, sejam estas pessoas ou veículos (atores). O simulador tem então de permitir definir cenários com base em mapas de estrada, e de forma automática gerar tráfego pedestre e veicular respeitando estas características, de forma a que cada ator tenha uma mobilidade independente.

Este simulador tem a capacidade de simular a comunicação oportunística entre automóveis, autocarros, pessoas, etc., quando estes elementos estão suficientemente próximos entre si. Esta forma de comunicação enquadra-se em várias áreas de aplicação como sistemas de prevenção de acidentes de viação, gestão de situações de emergência ou até controlo de tráfego rodoviário.

Este simulador tem ainda a característica de ser distribuído, permitindo assim que a as várias partes da simulação possam ser processadas em máquinas diferentes em simultâneo, permitindo assim aumentar o poder de processamento.

1.2 OBJETIVOS

Uma vez inteirado do enquadramento em que está inserido o simulador de movimento e comunicação entre atores num cenário urbano, pode-se assim definir os seguintes objetivos gerais para a dissertação:

1. Desenho de um conjunto de componentes que permitam simular sistemas baseados em comunicações oportunísticas.
2. Adaptar o código fonte da versão atual do simulador, para que este possa suportar a simulação das comunicações oportunísticas entre atores.
3. Implementar e integrar no simulador a simulação dos sistemas baseados em comunicações oportunísticas, neste caso de trabalho em concreto a tecnologia Bluetooth.
4. Testar e avaliar o desempenho do simulador face ás comunicações oportunísticas inseridas.

1.3 ESTRUTURA DO DOCUMENTO

Esta dissertação encontra-se dividida em oito capítulos. Neste primeiro capitulo é feito um enquadramento e são esclarecidos os objetivos desta dissertação.

No segundo capitulo como forma introdutória, é feita uma contextualização tecnológica das tecnologias que abrangem esta dissertação, residindo na tecnologia Bluetooth.

No capitulo número três "Simuladores de Mobilidade", é feito um estudo a diversos simuladores de mobilidade existentes, com vista a captar técnicas e ideias a aplicar neste simulador.

No capitulo quarto, é explicado todo o funcionamento do simulador, onde é feita uma descrição da arquitetura geral, e explicados todos os blocos do simulador.

O capitulo cinco é dedicado à comunicação entre atores neste simulador. São explicadas todas as técnicas e métodos implementados na construção destas novas funcionalidades do simulador.

O sexto capitulo foca-se na implementação da simulação da tecnologia Bluetooth. É explicada a arquitetura desenhada e todos os seus blocos subjacentes construídos.

No sétimo capitulo são mostrados os testes efetuados ao simulador antes e após a implementação das novas funcionalidades, onde são também apresentados os respetivos resultados obtidos.

O oitavo e último capitulo, apresenta as conclusões do projeto e ainda o trabalho que poderá ser feito no futuro neste simulador.

2

CONTEXTO TECNOLÓGICO

Este capitulo tem como finalidade expor os aspetos técnicos da tecnologia envolvida nesta dissertação, nomeadamente a tecnologia de comunicação Bluetooth.

2.1 BLUETOOTH

O Bluetooth[2] é uma tecnologia sem fios criada para comunicação em curtas distâncias com um baixo custo e alta operabilidade. Foi a Ericsson, em 1994, que começou a estudar esta tecnologia, projetada para permitir a comunicação entre telemóveis e acessórios, utilizando sinais de rádio, permitindo assim um baixo custo ao invés dos tradicionais cabos. Rapidamente outras empresas mostraram o seu interesse pela tecnologia e começaram a ajudar na pesquisa desta. Nokia, Ericsson, Intel, IBM e Toshiba fizeram assim nascer o consórcio Bluetooth SIG (Special Interest Group) para que fossem desenvolvidos padrões que garantissem o uso e a interoperabilidade da tecnologia nos mais variados dispositivos. Atualmente este consórcio conta com mais de 14.000 empresas como membros que promovem e desenvolvem a tecnologia. O IEEE (Institute of Electrical and Electronics Engineers) definiu a tecnologia como o *standard* 802.15, relativamente às PAN (Personal Area Networks).

Esta tecnologia veio assim abrir portas para a comunicação sem fios a curta distância de forma simples e eficaz. Obteve assim sucesso mundial atingindo diversos dispositivos, tais como telemóveis, computadores ou auriculares. Esta tecnologia oferece ainda mais possibilidades tais como a divulgação de conteúdos multimédia, por exemplo com fins publicitários.

Embora seja uma tecnologia cujo meio de comunicação são as ondas de radiofrequência, tem uma resistência razoável no que diz respeito às interferências do meio de comunicação. Outra grande vantagem é o facto de não necessitar que os dispositivos envolvidos na rede estejam em linha de vista tal como acontece com os infravermelhos. Permite também que essa mesma rede seja integrada por vários dispositivos em simultâneo.

Os vários protocolos da tecnologia Bluetooth foram desenhados de forma a poderem suportar comunicação de voz e dados, e ainda outros protocolos como o TCP. Tal como estudado e comparado em [1], outro ponto forte desta tecnologia é o baixo consumo de energia, que consegue ser bastante inferior quando comparado por exemplo com a tecnologia Wi-Fi[6] ou GSM-Edge[5].

Contudo esta tecnologia também apresenta alguns pontos fracos, como a taxa de transmissão que não é muito elevada quando comparada, por exemplo, à norma 802.11.g. O curto alcance (habitualmente até 10 metros) é também uma desvantagem comparando com outras tecnologias embora seja uma característica desta tecnologia.

2.1.1 *Rede, Piconets e Scatternets*

Um piconet é nada mais nada menos do que um dispositivo Slave ou um conjunto de dispositivos Slaves, a operarem conjuntamente com um dispositivo Master comum, tal como exemplifica a Figura 1.

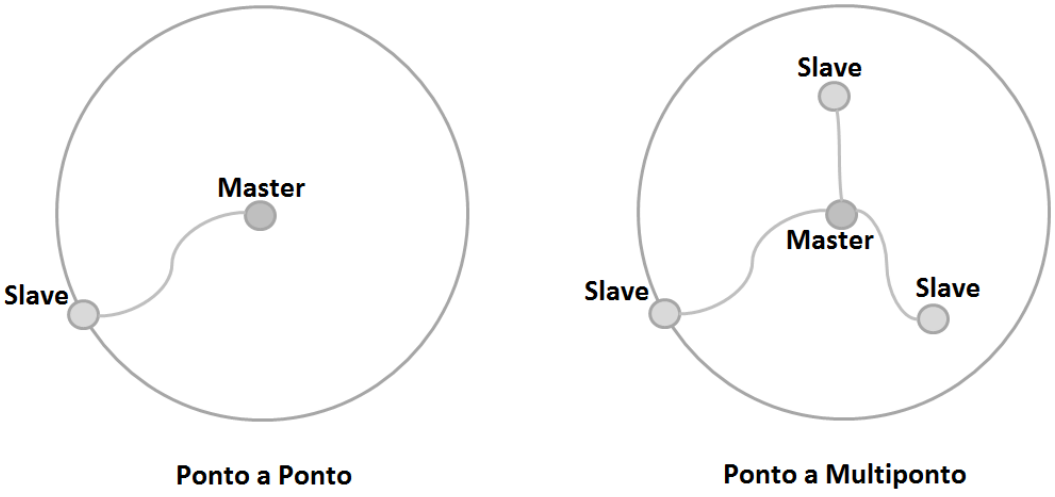


Figura 1: Bluetooth: Piconet ponto a ponto e ponto a multiponto

Podemos ver então na Figura 1 duas possibilidades de uma piconet. Na piconet da esquerda vemos apenas um Slave com uma ligação ponto a ponto ao Master. Na piconet da direita vemos três Slaves ligados ao Master, que são as únicas ligações possíveis, pois os Slaves numa piconet não estabelecem ligações diretas entre si, apenas estabelecem ligações diretas

com o Master.

A especificação Bluetooth limita o número de Slaves numa piconet, sendo sete Slaves o limite, onde cada Slave apenas comunica com o Master partilhado. Contudo, é possível criar uma rede de maior cobertura, permitindo agregar um maior número de Slaves pela agregação de piconets numa scatternet, onde alguns dispositivos Bluetooth são membros de mais de que uma piconet, como exemplifica a Figura 2.

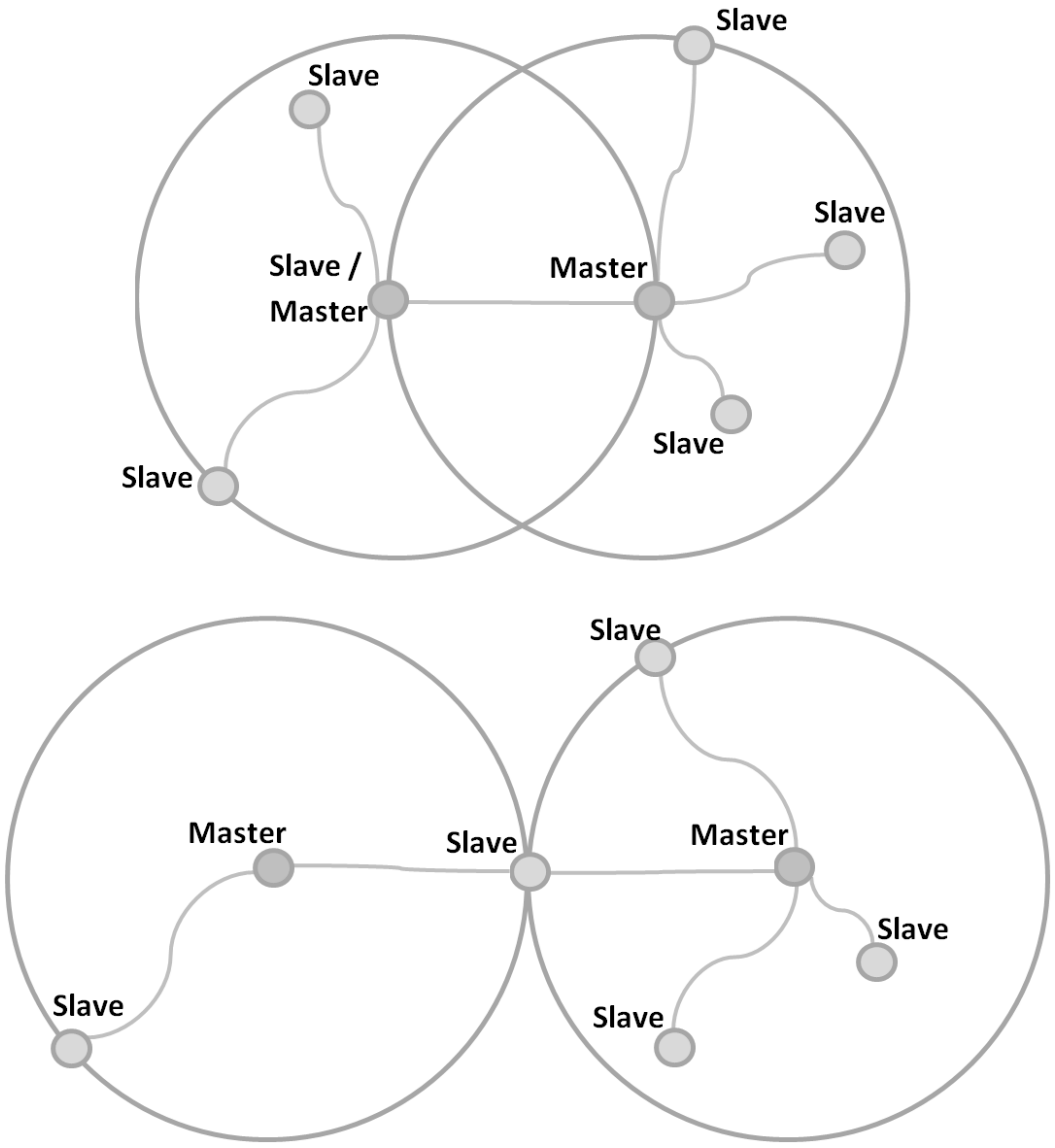


Figura 2: Bluetooth: Scatternets [Adaptada de [3]]

Quando um dispositivo se encontra em mais de que uma piconet, este tem que partilhar o seu tempo, dispendendo alguns slots numa piconet e

outros slots noutra. No exemplo de cima da Figura 2 podemos ver uma scatternet onde um dispositivo é Slave numa piconet e Master noutra. No exemplo de baixo, podemos ver o caso em que um dispositivo é Slave em duas piconets diferentes. Pode-se também concluir que todos os dispositivos com o mesmo Master, têm de estar portanto na mesma piconet.

Perante estas caraterísticas, pode-se também dizer que entre as várias fontes possíveis de interferências, a que apresenta maior fonte de interferência para o Bluetooth, é claramente outro dispositivo Bluetooth. Visto que vários dispositivos partilham uma piconet, estes devem assim sincronizar-se para evitar que outras piconets dessincronizadas na área possam colidir na mesma frequência, o que provocará perda de dados e a consequente retransmissão, ou no caso de voz, a simples perda de informação. Concluindo assim, que quanto maior for o número de piconets na área, maior a probabilidade de existirem colisões e assim piorar a qualidade da comunicação. Tal situação aplica-se também no caso das scatternets, pois estas não coordenam entre si os saltos em frequência.

2.1.2 *Arquitetura*

Uma característica da especificação Bluetooth é que esta permite que dispositivos de diferentes marcas e origens possam trabalhar entre si. Com este intuito, tornou-se necessário, para além de definir o sistema de rádio, definir também uma pilha de software para permitir às aplicações encontrarem outros dispositivos Bluetooth na área, descobrir quais os serviços que estes oferecem e usá-los.

Foi assim construída uma arquitetura protocolar em camadas como mostra a Figura 3, sendo esta constituída por três grupos lógicos:

- Grupo de protocolos de transporte
- Grupo de protocolos de middlleware
- Grupo de aplicação

Comparando o modelo protocolar da tecnologia Bluetooth com o modelo protocolar *standard* de referencia do modelo OSI, podemos concluir que não combinam de forma exata, tal como mostra a Figura 4.

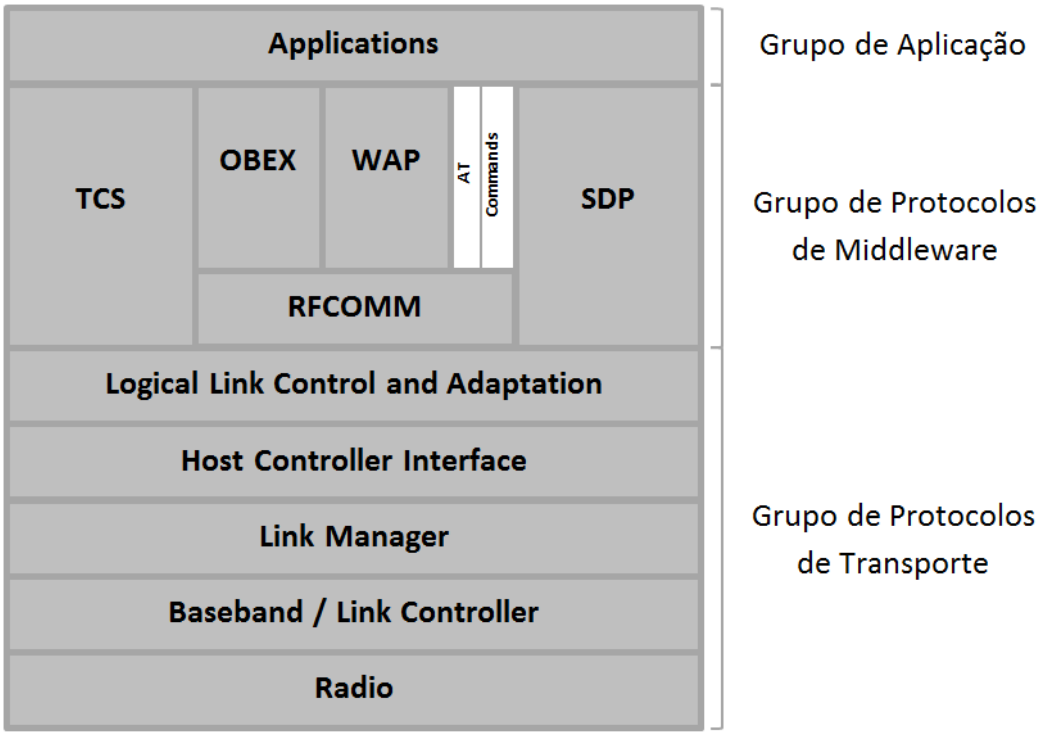


Figura 3: Bluetooth: Arquitetura Protocolar de Camadas [Adaptada de [3]]

Sabendo que o modelo OSI é considerado um modelo de referência na boa partição da pilha protocolar, poderemos assim fazer uma comparação com o modelo protocolar do Bluetooth, destacando dessa forma a divisão de responsabilidades da pilha protocolar.

A camada física é a responsável pelo interface elétrico para o meio de comunicação, incluindo a modulação e codificação do canal.

A camada de ligação é responsável pela reconstrução de pacotes e controle de erros de uma ligação, e como tal tem tarefas diretamente relacionadas com a camada Link Controller e com o controlo final da camada Baseband, incluindo controlo de erros e correção.

A camada de rede é responsável pela transferência de dados através da rede, independentemente do meio e da topologia de rede. Dessa forma, a parte mais "alta"da camada Link Controller, estabelece e mantém múltiplas ligações e cobre também grande parte das tarefas da camada Link Manager.

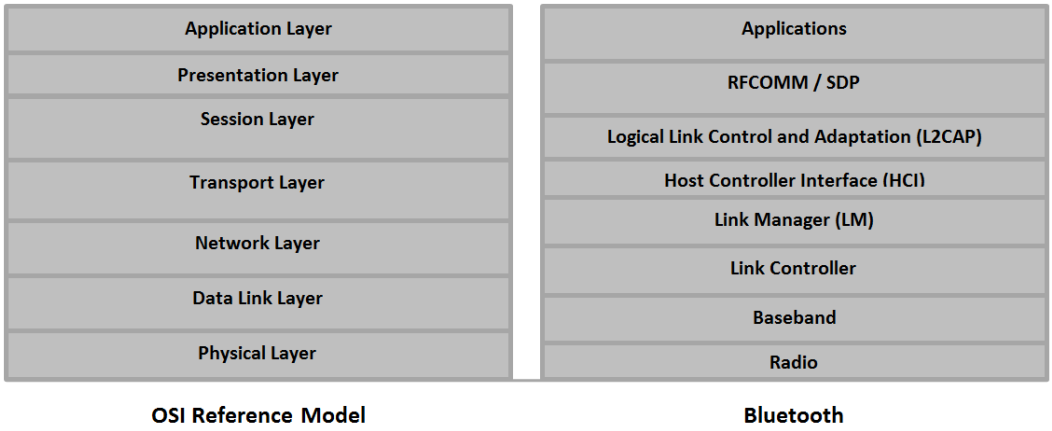


Figura 4: Arquitetura Protocolar Bluetooth vs. OSI [Adaptada de [3]]

A camada de transporte é responsável pela segurança e multiplexagem da transferência de dados através da rede para o nível disponibilizado pela aplicação. E está assim diretamente relacionada com a parte mais alta da camada Link Manager e cobre também a camada Host Controller Interface que dispõe dos mecanismos de transporte.

A camada de sessão é a que oferece a gestão e os serviços de controlo de fluxo de dados, os quais são cobertos pela camada L2CAP e pela parte mais baixa da camada RFCOMM/SDP.

A camada de apresentação disponibiliza uma apresentação comum aos dados da camada de aplicação, adicionando serviços às unidades de dados, sendo esta a principal função da camada RFCOMM/SDP.

Finalmente a camada de aplicação é a responsável por gerir as comunicações entre as aplicações nos dispositivos.

2.1.2.1 *Radio*

Sendo o Bluetooth uma tecnologia destinada ao funcionamento em qualquer parte do globo, foi assim necessário adotar uma frequência de rádio aberta padrão. A melhor solução encontrada foi usar a banda ISM (Industrial, Scientific, Medical), e foi assim definido operar na gama de 2.4GHz a 2.48GHz.

Como a banda ISM é aberta, esta pode ser utilizada por qualquer sistema de comunicação, e é então fulcral garantir que o sinal Bluetooth não gera nem sofre interferências. Para tal foi adotado o esquema de comu-

nicação FHSS (Frequency Hopping – Spread Spectrum), pois desta forma cada frequência é dividida em vários canais. O dispositivo que estabelece a conexão vai saltando de canal em canal de forma muito rápida (Frequency Hopping), e isto faz com que a largura de banda da frequência seja muito pequena, diminuindo as hipóteses de haver interferências. Dependendo do país o Bluetooth utiliza de 23 a 79 frequências na banda ISM, distanciadas por 1MHz entre si.

Visto que a tecnologia Bluetooth permite a transmissão de dados em full-duplex, o mecanismo de Frequency Hopping é usado tanto para a emissão como para a recepção de pacotes, usando esquemas de divisão temporal Time-Division-Duplex (TDM). A transmissão é assim alternada entre slots de transmissão e slots de recepção com 625 µs como intervalo de tempo, tal como mostra a Figura 5. Dessa forma cada salto de frequência deve ser ocupado por um slot, perfazendo assim um total de 1600 saltos por segundo.

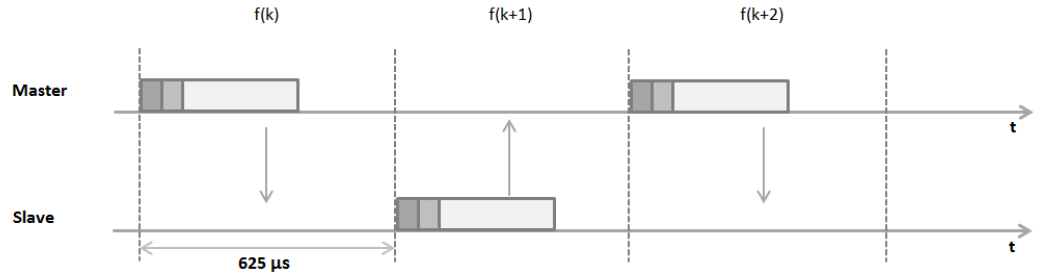


Figura 5: Bluetooth: Time Division Multiplex

Se os dispositivos saltam para novas frequências após cada pacote, eles devem combinar entre si a sequência de frequências a usar. Os dispositivos Bluetooth podem operar em dois diferentes modos, Master ou Slave. É o Master que define a sequencia de saltos em frequência, portanto, é tarefa do Slave sincronizar-se com o Master em tempo e em frequência, seguindo a sequência de saltos em frequência.

Todos os dispositivos Bluetooth possuem um endereço Bluetooth único e um relógio Bluetooth. Estas características são usadas pela camada Baseband da pilha protocolar, que dispõe de algoritmos que calculam a sequência de saltos em frequência, com base no endereço e relógio Bluetooth. Desta forma, quando um Slave pretende conetar-se a um Master, recebe o endereço e relógio do mesmo, e podem assim usar esses dados para cal-

cular a sequência de saltos em frequência. Desta forma podemos afirmar que como os Slaves usam sempre o endereço e relógio do Master, todos se sincronizam à sequência de saltos em frequência deste.

Para além de controlar a sequência de saltos em frequência, o Master tem ainda a possibilidade de controlar quando os dispositivos têm permissão para transmitir. O Master permite que os dispositivos transmitam pela alocação de slots para tráfego de voz ou tráfego de dados. Nos slots para tráfego de dados, os Slaves podem transmitir apenas em resposta de uma transmissão recebida do Master. Quanto aos slots de tráfego de voz, é exigido aos Slaves que transmitam regularmente em slots reservados, quer sejam em resposta ou não a uma transmissão recebida do Master.

O Master tem ainda a função de controlar como a largura de banda total disponível é dividida pelos Slaves, tendo assim de decidir quando e com que frequência pode comunicar cada Slave. O número de slots de tempo que cada dispositivo recebe, depende dos seus requisitos para a transferência de dados.

Originalmente a modulação usada pelo Bluetooth foi a modulação GFSK[4], mas posteriormente com a versão 2.0+EDR, começaram também a ser usadas as modulações $\pi/4$-DQPSK[17] e a 8DPSK[17], que podem ser então usadas com dispositivos compatíveis.

Quando os dispositivos usam a modulação GFSK operam no modo BR (Basic Rate) é possível obter um débito de 1 Mbit/s.

Quando usados dispositivos compatíveis com a modulação $\pi/4$-DQPSK ou a 8DPSK, pode ser usado o modo EDR (Enhanced Data Rate) permitindo assim 2 Mbits/s ou 3 Mbits/s respetivamente. É ainda possível combinar estes dois modos sendo este sistema classificado de "BR/EDR radio".

O protocolo Bluetooth foi desenhado para responder a comunicações de curto alcance, contudo existem 3 classes com diferentes características de emissão tal como podemos ver na Tabela 1.

As classes de potência permitem que os dispositivos Bluetooth possam ter conexões com diferentes alcances. A classe 2 é hoje em dia a classe

Classe	Potência Máxima	Alcance
Classe 1	100mW (20dBm)	100 metros
Classe 2	2.5mW (4dBm)	10 metros
Classe 3	1mW (0dBm)	1 metro

Tabela 1: Bluetooth: Classes de Potência

mais adotada, permitindo estabelecer conexões até aproximadamente 10 metros de distância, com um baixo custo e baixa potencia de comunicação, sendo portanto uma excelente alternativa para a substituição de cabos.

Obviamente, quanto maior a potencia emitida maior o alcance. A classe 1 é a classe com maior potência emitida, podendo assim o alcance chegar até aproximadamente 100 metros de distância. Ainda assim, é considerada uma potência baixa e portanto livre de preocupações para a saúde. Quando comparada com outros métodos de comunicação como o GSM 850/900, podemos ver que a classe de potência máxima do Bluetooth é ainda assim 20 vezes inferior à máxima potencia emitida pelo GSM 850/900.

Devido à saturação do sinal quando o receptor se encontra muito próximo do emissor, existe uma distância mínima necessária entre emissor e receptor, sendo esta usualmente de 10 centímetros.

2.1.2.2 *Baseband*

A camada baseband é a camada responsável pela codificação e descodificação do canal, e é ainda a camada responsável pelas operações de controlo de baixo nível de temporização e gestão de ligações. Cabe assim a esta camada definir os tipos de pacotes, procedimentos de processamento de pacotes, métodos de deteção de erro, criptografia, transmissão e retransmissão de pacotes, métodos de descoberta de outros dispositivos, métodos de ligação a outros dispositivos e ainda a definição dos papeis de Master e Slave.

2.1.2.3 *Ligação SCO e ACL*

Quando é estabelecida uma ligação entre um Master e um Slave, existem dois tipos possíveis de ligação que irão definir características diferentes dos pacotes de dados dessa ligação.

O primeiro tipo é o SCO (Synchronous Connection Oriented). Estas ligações são caraterizadas pela atribuição periódica de um slot de tempo a um dispositivo, permitindo dessa forma que os pacotes tenham prioridade. Estas ligações são utilizadas nas ligações de voz, que requerem transmissões de dados rápidas e consistentes. Assim sendo, um dispositivo que estabelece uma ligação SCO tem slots de tempo reservados para si, e portanto prioritários pois serão processados primeiro que os slots ACL.

O outro tipo de ligação possível é o ACL (Asynchronous Connection-Less). Este tipo de ligação não prevê qualquer tipo de reserva de slots ao contrário do tipo de ligação SCO. Contudo as ligações ACL visto não serem destinadas a voz, mas sim a transferência de dados, permitem que sejam enviados pacotes com tamanho variável, podendo estes ocupar 1, 3 ou 5 slots de tempo.

2.1.2.4 *Estrutura dos Pacotes*

Cada pacote Bluetooth tem uma estrutura padrão dividida em três partes: access code, header e payload tal como mostra a Figura 6. Contudo existem várias variações na construção dos pacotes, podendo estes ser divididos em três tipos: controlo, SCO e ACL.

Figura 6: Bluetooth: Estrutura do Pacote

O access code é geralmente usado para que seja identificada a piconet e seja feita a sincronização. É desta forma que um Slave deteta a presença de um pacote e o aceita ou não pela comparação do access code do pacote e da sua cópia guardada do access code do Master. Este método é também uma forma simples de identificar a presença de diferentes Masters na área, e ajudando também na tentativa de criação de uma scaternet.

O header de um pacote Bluetooth contém a informação de controlo associada ao pacote e à ligação. Inclui seis campos de informação que prefazem um total de 18 bits, os quais estão protegidos pelo mecanismo FEC 1/3 (Forward Error Correction), o que significa que os dados são replicados três vezes, daí o total ser 54 bits e não apenas 18 bits.

A parte final de payload é a parte onde circulam os dados, contudo é aqui onde existem diferenças perante os pacotes do tipo SCO e do tipo ACL. No caso dos pacotes ACL, como mostra a Figura 7 o payload pode ter até 2744 bits de tamanho, enquanto que nos pacotes SCO, o tamanho é fixo e corresponde a 240 bits, sem que exista qualquer cabeçalho de payload ou controlo de erros.

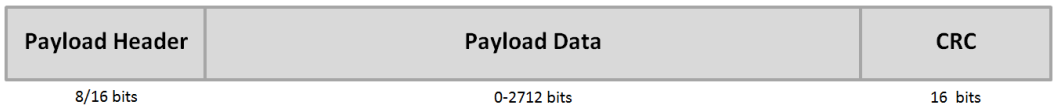


Figura 7: Bluetooth: Estrutura do Payload ACL

2.1.2.5 *The Link Controller*

Esta camada, tal como o nome aponta, é a camada responsável pela gestão/manutenção das ligações.

Num determinado instante de tempo, um dispositivo Bluetooth encontra-se num determinado estado. Podemos ver na Figura 8 o diagrama de estado de um dispositivo Bluetooth com todos os estados possíveis em que este se poderá encontrar.

- Standby
  O estado de standby ocorre quando um dispositivo se encontra inativo, portanto não existindo qualquer tipo de troca de informação, e o rádio encontra-se desligado. Dessa forma o dispositivo não descobrirá qualquer access code. Este modo é normalmente usado para poupança de energia.

- Inquiry
  Inquiry é o processo onde um dispositivo Bluetooth procura outros dispositivos Bluetooth ativos na área. O serviço SDP (Service Discovery Protocol) permite que uma dispositivo obtenha uma lista de

outros dispositivos na área, com quem possivelmente se poderá ligar. Um Inquiry Respose é uma resposta a um Inquiry.

- Inquiry Scan
  Para que no modo inquiry os dispositivos consigam encontrar outros dispositivos na área, é necessário que estes estejam em modo inquiry scan, fazendo assim leituras de inquiries e respondendo aos mesmos.

- Page
  Para estabelecer uma conexão, o dispositivo que será o Master é responsável por tratar dos procedimentos de paging. O Master começa então por entrar no modo de page onde irá transmitir mensagens de paging diretamente para o dispositivo Slave, usando para isso o access code obtido no processo de inquiry.

- Page Scan
  Neste modo tal como o inquiry scan, o dispositivo irá fazer de forma periódica leituras de paging, permitindo assim o estabelecimento de uma conexão com um dispositivo que lhe enviou um pedido.

- Connection - Active
  Quando uma comunicação é bem sucedida entre um Master e um Slave, este novo dispositivo entra no modo Connection - Active, e recebe então um endereço que o identifica na piconet. Ao entrar para este estado, o Slave tem assim de mudar para o relógio do Master, e mudar também para os saltos em frequência fornecidos pelo Master. Neste estado poderão ser trocados inúmeros pacotes de dados, contudo periodicamente o dispositivo poderá mudar-se para um modo de baixo consumo de energia.

- Connection - Hold
  No modo Connection - Hold o dispositivo deixa de permitir ligações ACL por um período de tempo, para dar largura de banda a outras operações tais como scanning, paging ou inquiry.

- Connection - Sniff
  Neste modo é dado a um Slave um valor pré-definido de slots de tempo e periodicidade para que este faça então uma escuta periódica de pacotes. Se num dos momentos em que o Slave vai procurar por pacotes e recebe um destinado a si, pode então continuar com a troca continuada de dados até que haja um time out, e assim volta ao estado anterior de leitura periódica.

- Connection - Park
  Quando um dispositivo Slave não tem dados a serem enviados ou recebidos, o Master pode optar por intuí-lo a entrar no modo Connection - Park. Neste modo o dispositivo Slave passa a escutar pacotes apenas ocasionalmente, e na maior parte do tempo poderá então entrar em modo Low Power Sleep.

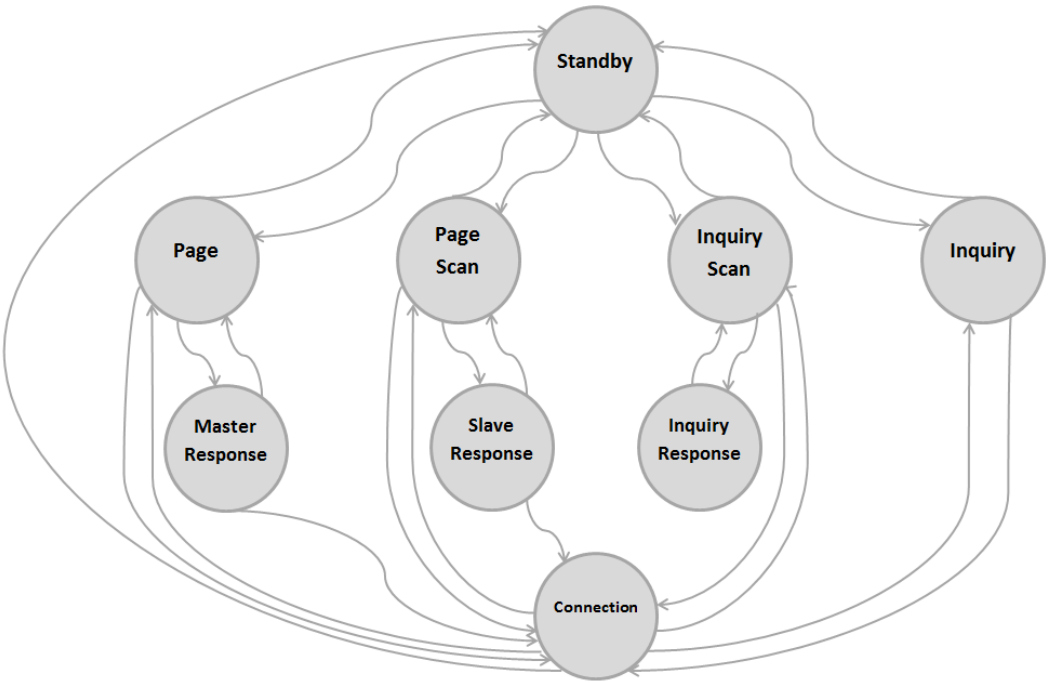


Figura 8: Bluetooth: Diagrama de Estado [Adaptada de [7]]

2.1.2.6 *The Link Manager*

A camada Link Manager é a camada responsável pela tradução dos comandos HCI (Host Controller Interface) em comandos ao nível da baseband, fazendo a gestão das seguintes operações.

- Adicionar Slaves a uma piconet e fazer a alocação do endereço de membro ativo.
- Cortar ligações para retirar Slaves de uma piconet.
- Configuração de ligações, incluindo controlo de trocas entre Master e Slaves, onde ambos os dispositivos devem fazer a mudança em simultâneo.
- Estabelecer ligações ACL e SCO.
- Controlo de modos de teste.

2.1.2.7 *The Host Controller Interface*

Alguns sistemas Bluetooth têm a camada baseband e a camada link manager num mesmo processador e as camadas mais altas como a L2CAP, SDP, RFCOMM ou as aplicações, a correr num processador separado. Como exemplo disso uma placa Bluetooth PCMCIA, poderá implementar este sistema com as camadas link manager e baseband a correrem na placa PCMCIA e as camadas mais altas a correrem no processador do computador.

Nestes sistemas onde as camadas mais altas estão a correr no processador do dispositivo e as camadas mais baixas a correr no dispositivo Bluetooth, é necessário haver um interface entre as camadas altas e as camadas mais baixas, e é esta a função da camada HCI no protocolo Bluetooth, tal como podemos ver na Figura 9. Sendo este um interface standard, torna-se assim possível fazer uma correspondência entre camadas de baixo nível e alto nível, permitindo assim por exemplo que o software de um computador possa dessa forma ser usado com placas PCMCIA de diferentes produtores.

Existem várias razões para que assim seja, tais como:

- Dispositivos como computadores, separam as capacidades de gestão das altas camadas, permitindo que os dispositivos Bluetooth usem menos memória, e tenham também menos processamento para reduzir custos.
- Um dispositivo poderá adormecer e ser acordado, pela chegada de uma conexão Bluetooth.

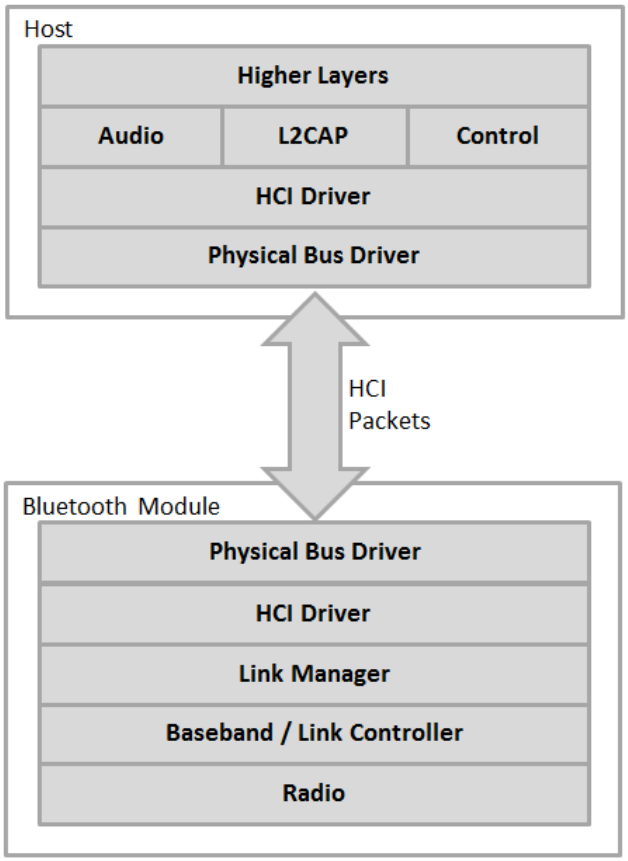


Figura 9: Bluetooth: HCI na pilha protocolar [Adaptada de [3]]

- A camada HCI é útil para a realização de testes e para fazer uma aceitação de dispositivos Bluetooth.

2.1.2.8 *Logical Link Control and Adaptation Protocol*

A camada Logical Link Control and Adaptation Protocol, mais conhecida por L2CAP, é responsável por receber dados provenientes das camadas superiores e envia-los para as camadas mais baixas na pilha. De referir que não há troca de pacotes de áudio através desta camada, apenas se estes forem via VOIP.

São por isso funções da camada L2CAP:

- Fazer a multiplexagem entre diferentes protocolos de camadas de nível superior, permitindo que estes partilhem as ligações de baixo nível.

- Fazer a segmentação e reconstrução de pacotes, para que seja possível a transferência de pacotes maiores de que suportados nas camadas mais baixas.
- Gestão de grupos, permitindo transmissões de um sentido para grupos de outros dispositivos.
- Fazer a gestão de QoS (Quality of Service) para os protocolos das camadas superiores.

2.1.2.9 *RFCOMM*

Esta camada tem a função de emular uma porta serie sobre a camada L2CAP, permitindo desta forma suportar todas as aplicações que usem comunicação via porta serie. A RFCOMM permite ainda a existência de conexões concorrentes, para isso confiando na capacidade de multiplexing da camada L2CAP, permitindo até 60 portas abertas, usando para isso um DLCI (Data Link Connection Identifier) que identifica a ligação entre o cliente e a aplicação.

RFCOMM é um protocolo de transporte com fragmentação e multiplexação simples e confiável, sendo ainda o responsável pelo controlo de variáveis como:

- Modem Status - RTS/CTS, DSR/DTR, DCD, e ring.
- Remote line status - Break, overrun, parity.
- Remote Port Settings - Baud Rate, Parity, No of data bits.
- Parameter Negotiation (frame size).

2.1.3 *Procura de Dispositivos*

Sendo o Bluetooth uma tecnologia sem fios, não existe assim a necessidade de estabelecer conexões físicas via cabo. Dessa forma torna-se complicado saber que dispositivos estão próximos e quais os serviços que oferecem. Para resolver esta problema a tecnologia Bluetooth dispõe então de mecanismos de pesquisa e paginação.

Quando um dispositivo Bluetooth quer estabelecer uma ligação com outro dispositivo Bluetooth, o primeiro passo a dar é descobrir quais os dispositivos que estão ativos na área. Para efetuar esta ação, o dispositivo que

pretende obter uma ligação a outro dispositivo, deve então enviar uma série de pacotes *inquiry* para o meio de transmissão, esperando que os dispositivos na área o recebam, e lhe respondam. Quando eventualmente outro dispositivo responde ao inquiry, essa resposta é nada mais nada menos do que um pacote FHS (Frequency Hop Synchronisation), tal como demostra a Figura 10. Este pacote contem toda a informação necessária para que o dispositivo consiga então criar uma conexão com o dispositivo alvo, tal como a chave de sincronização ou o instante do relógio Bluetooth.

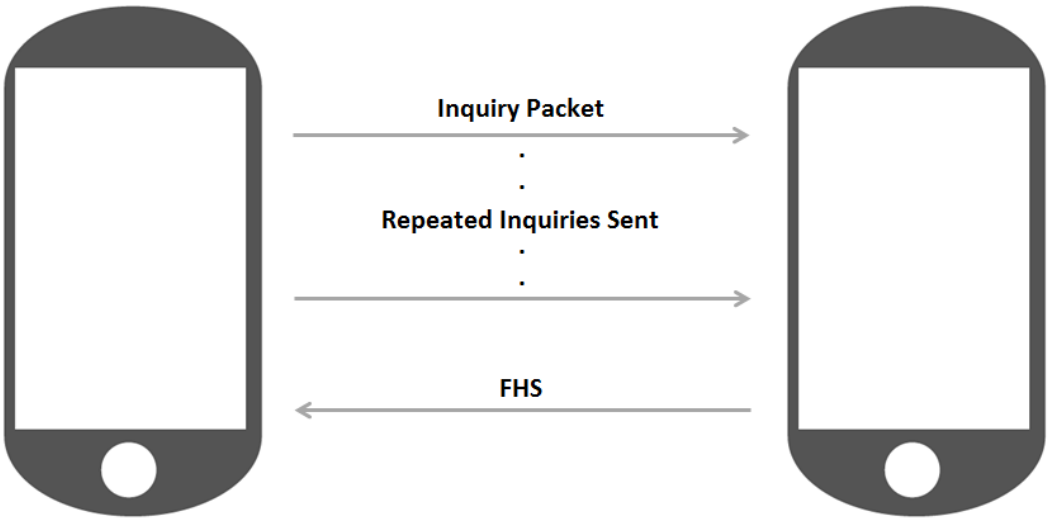


Figura 10: Bluetooth: Descoberta de outros dispositivos [Adaptada de [3]]

Assim sendo, todos os dispositivos Bluetooth ativos na área irão responder ao inquiry com um pacote FHS. Dessa forma, se na área houver mais do que um dispositivo Bluetooth ativo e à espera de inquiries, irá assim ser obtida uma lista de dispositivos disponíveis, e cabe então à aplicação decidir o que fazer com a lista, sendo uma das possibilidades apresentá-la ao utilizador para que este decida com que dispositivo deseja estabelecer a conexão.

2.1.4 *Procura de Serviços*

Para descobrir se um determinado dispositivo suporta um determinado serviço, depois dos dispositivos estabelecerem uma ligação é usado o SDP (Service Discovery Protocol). Este protocolo funciona da seguinte forma: após haver uma ligação estabelecida entre os dispositivos, o dispositivo que pretende descobrir os serviços disponibilizados pelo outro dispositivo faz uma paginação ao dispositivo alvo. Para isso, este tem que estar à

espera de pedidos de paginação, e dessa forma após responder ao pedido pode então ser estabelecida uma ligação ACL entre ambos, para que estes possam trocar dados.

Após ter sido estabelecida uma conexão ACL, pode ser então estabelecida uma conexão L2CAP, sendo esta a conexão que é usada quando existe troca de dados entre dispositivos Bluetooth. L2CAP permite que diferentes protocolos e serviços usem a mesma ligação ACL, distinguindo os diferentes serviços e protocolos, través do PSM (Protocol and Service Multiplexor). O PSM é diferente em cada serviço ou protocolo, tendo neste caso especifico de pesquisa de serviços o valor PSM = 0x0001. Com o canal L2CAP o dispositivo consegue então estabelecer uma conexão com o servidor SDP do dispositivo alvo, permitindo assim que seja obtida toda a informação sobre os serviços disponibilizados.

Tal como mostra a Figura 11, imaginemos que um dispositivo Bluetooth A pretende saber se o dispositivo Bluetooth B dispõe de algum serviço de DUN (Dial Up Network). Após estes dois dispositivos estabelecerem uma conexão, o dispositivo A faz então uma paginação ao dispositivo B, e caso este esteja a fazer leituras de pedidos de paginação, responde ao dispositivo A permitindo assim estabelecer uma conexão ACL. Com a conexão ACL, torna-se possível que através da L2CAP o dispositivo A consiga aceder ao servidor SDP do dispositivo B, e pedir assim que este lhe envie toda a informação relacionada com o serviço de DUN.

Quando o dispositivo A tiver obtido a informação pedida, deve terminar a conexão com o dispositivo B, visto que o número de conexões ativas é limitado, e este pode querer repetir o mesmo processo com outros dispositivos Bluetooth ativos na área, para além de permitir uma melhor utilização da bateria.

Com a informação recebida do dispositivo B, cabe à aplicação decidir o que fazer com a informação. Esta tanto pode apresentar a informação ao utilizador para que este decida o que fazer, tal como decidir por ela própria. De qualquer das formas, a informação recebida contem toda a informação necessária para que o dispositivo A use o serviço DUN do dispositivo B.

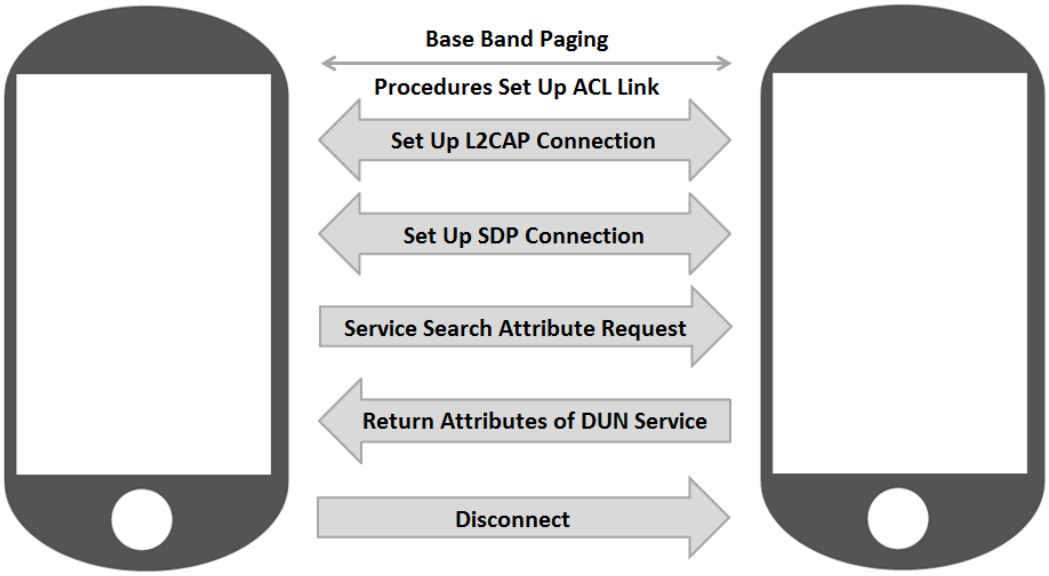


Figura 11: Bluetooth: Processo de Procura de Serviços [Adaptada de [3]]

2.1.5 *Uso de Serviços*

Depois de um dispositivo saber que um serviço é disponibilizado por outro, este pode então fazer uma conexão com esse dispositivo e usar o serviço em causa. Em primeiro lugar, tal como acontece no processo de pesquisa de serviços, após o processo de paginação é estabelecida uma ligação ACL.

Neste caso, onde pretendemos usar um serviço, o protocolo de ligação poderá ser estabelecido tendo em conta requisitos particulares de qualidade de serviço. Para tal, a aplicação a correr no dispositivo que pretende usar o serviço, deve configurar a ligação para que tais requisitos possam ser alcançados. Dessa forma, a aplicação deve realizar esses pedidos recorrendo ao Host Controller Interface, e de seguida o Link Management Protocol configura a ligação.

Uma vez que a ligação ACL se encontra estabelecida e configurada de forma a responder aos requisitos, é feita uma ligação L2CAP. Imaginando o exemplo da secção anterior, em que um dispositivo A pretende usar o serviço de DUN do dispositivo B, neste caso específico este serviço usa RFCOMM, uma camada de emulação de RS-232[1], e desta forma a L2CAP usa o PSM para o RFCOMM, PSM=0x0003.

1 Vulgarmente conhecido como porta série.

Depois de estabelecida a ligação L2CAP, pode ser então criada a ligação RFCOMM. A cada protocolo ou serviço é atribuído um determinado número de canal, o qual é fornecido no processo de descoberta de serviços, dessa forma o dispositivo A sabe qual o número de canal a usar ao estabelecer a conexão RFCOMM, para usar o serviço em causa.

Feito isto, pode finalmente ser estabelecida a ligação ao serviço DUN usando a ligação RFCOMM, e o dispositivo A pode então começar a usar o serviço DUN fornecido pelo dispositivo B. A Figura 12 mostra a sequência de todo este processo.

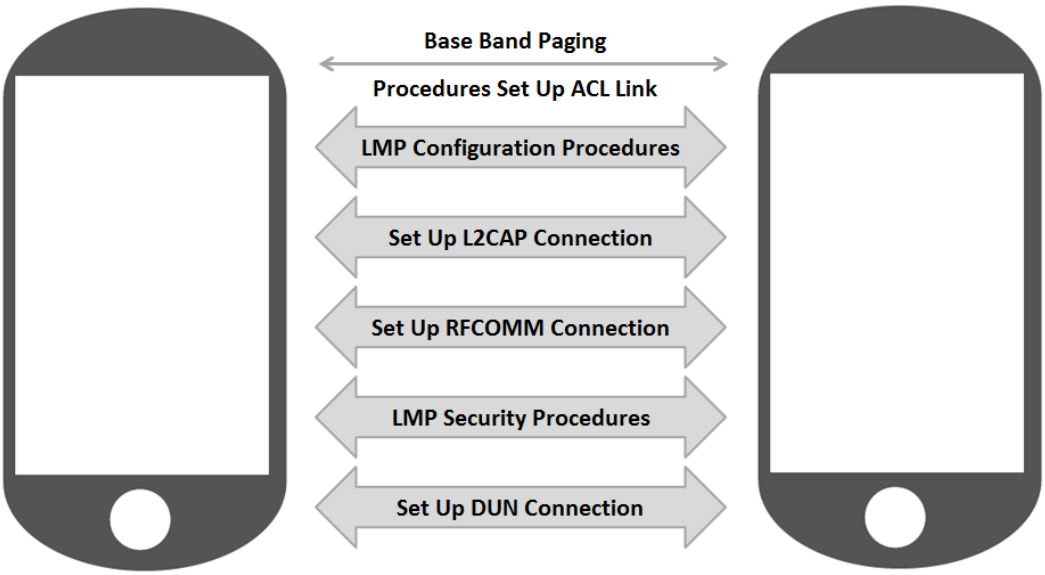


Figura 12: Bluetooth: Usar um Serviço [Adaptada de [3]]

3

SIMULADORES DE MOBILIDADE

Numa procura de ideias e técnicas a usar/implementar no simulador, foram estudados outros simuladores já existentes, os quais: Bonnmotion, Generic Mobility Simulation Framework, Simulation of Urban MObility, The One e VISSIM. Neste capitulo será então descrito o funcionamento dos diferentes simuladores.

3.1 BONNMOTION

Desenvolvido pelo grupo de Sistemas de Comunicação do Instituto de Computadores e Ciência IV da Universidade de Boon na Alemanha, o BonnMotion é um software open-source desenvolvido em Java. O software encontra-se atualmente na versão 2.1, lançado em Julho de 2013 e poderá ser legalmente descarregado na página online[1] do projeto.

3.1.1 *Geração de Cenários*

O BonnMotion permite a criação de vários tipos de cenários de mobilidade, dispondo assim de um grande leque de modelos de mobilidade implementados, sendo alguns dos quais:

- The Random Waypoint model
- The Manhattan Grid model
- Gauss-Markov models
  - The original Gauss-Markov model
  - The Gauss-Markov model
- Random Street
- Tactical Indoor Mobility Model
- Map-based Self-similar Least Action Walk

1 http://net.cs.uni-bonn.de/wg/cs/applications/bonnmotion/

- Steady-State Random Waypoint Model
- Random Direction Model

Para o uso de cada modelo, na chamada do simulador, é usado como parâmetro o nome do mesmo, e por ventura se se justificar o uso de outros parâmetros para a configuração do modelo.

3.1.2 *Formatos para exportação*

Por defeito, o BonnMotion guarda as movimentações feitas pelos nós num formato nó-por-linha, significando que cada linha do ficheiro corresponde a um nó e esta contém todos os *waypoints* pelo que o nó passou. Cada *waypoint* contém a posição x e y, e é ainda registado juntamente com o primeiro e último *waypoint*, o instante de tempo a que este corresponde.

Visto que o BonnMotion não dispõe nenhuma interface gráfica para apresentar os resultados produzidos, usando o ficheiro *.movements* resultado da simulação, com ferramentas como o GNUPlot, Excel, Matlab, SciLab, entre outras, podem-se desenhar os pontos num eixo cartesiano e dessa forma ter noção do movimento de um nó. A Figura 13 mostra os pontos onde o nó 1 esteve durante toda a simulação, apresentados pela aplicação GNUPlot.

Para além do formato *standard*, o BonnMotion permite também exportar os resultados para outros formatos de outros simuladores existentes, tais como:

- ns-2
- ns-3
- Glomosim / Qualnet
- IntervalFormat
- MiXiM
- ScenarioConverter
- CSVFile

O simulador permite ainda exportar para formato XML ou formato WiseML, que é o formato usado por exemplo pelo simulador de redes COOJA[13].

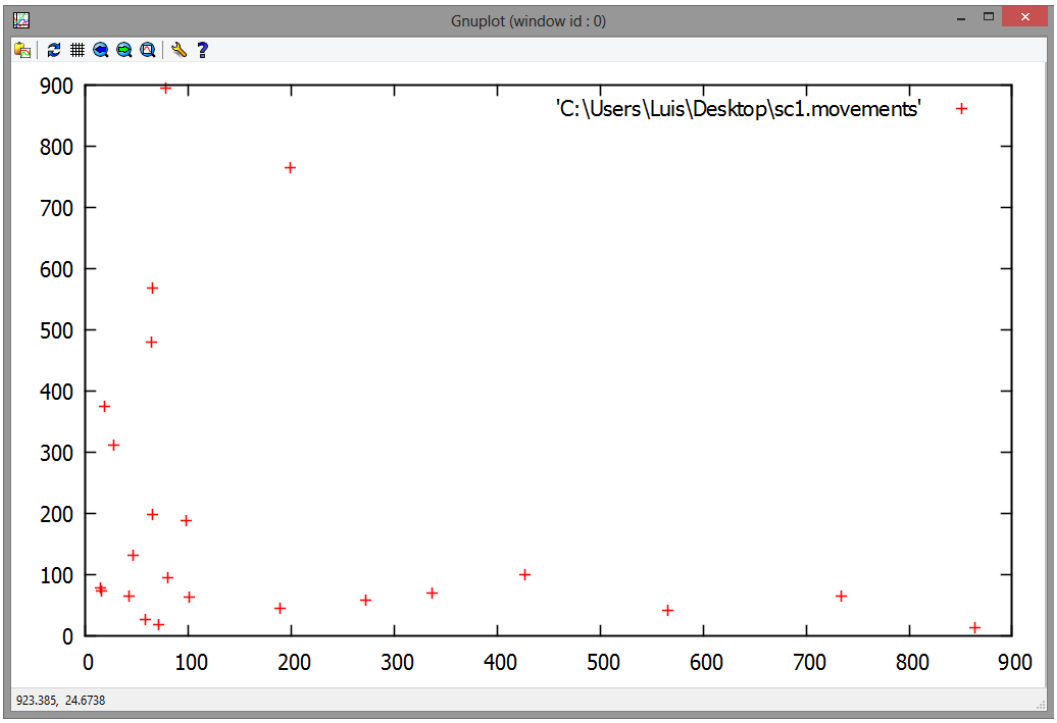


Figura 13: BonnMotion: Plot do Ficheiro Movements

3.1.3 *Estatísticas*

Este simulador dispõe de um módulo de análise da simulação, permitindo ao utilizador analisar e retirar dados estatísticos das simulações realizadas.

Existem dois modos de operação para o cálculo das estatísticas:

- Estatísticas de toda a simulação
- Estatísticas progressivas, por certos períodos de tempo

Por defeito a aplicação gera um ficheiro *.stats* que contém os dados estatísticos relativos à simulação, mostrado assim dados relativos da simulação, como por exemplo a velocidade média de todos os nós.

3.1.4 *Conclusões*

Como esta dissertação se foca na simulação das comunicações entre nós e na simulação da tecnologia *Bluetooth*, não será tido em conta este simulador como fonte de inspiração, pois este não dispões de mecanismo de troca de mensagens, focando-se apenas na mobilidade. Contudo, o módulo das

estatísticas é a um ponto a ter em atenção para uma futura evolução do BartUM.

3.2 GENERIC MOBILITY SIMULATION FRAMEWORK

Nascido em 2007 no grupo de sistemas de comunicações do Instituto Federal de Tecnologia de Zurique, o Generic Mobility Simulation Framework (GMSF) é uma ferramenta open-source desenvolvida em Java, que permite fazer a simulação e análise de mobilidade de nós em redes wireless, e foi desenhado para trabalhar em conjunto com simuladores de redes conhecidos. O simulador poderá ser legalmente descarregado na página online [2] do projeto.

Para a fazer a simulação de uma rede wireless, o GMSF necessita de saber a posição exata dos nós a cada instante de tempo, dessa forma foram implementados modelos de mobilidade, para criarem caminhos que podem ser exportados em diversos formatos. Para criar simulações ainda mais realistas, o GMSF dispõe ainda de alguns modelos de propagação de ondas eletro-magnéticas.

3.2.1 *Arquitetura*

O núcleo do GMSF centra-se no *Simulation Runtime Controller* e no *Data Storage Manager*, contudo a arquitetura é baseada em módulos, os quais são responsáveis por diferentes tarefas, tais como a definição das posições dos nós, geração de tráfego ou a formatação dos dados resultantes. A Figura 14 mostra a arquitetura dos componentes do simulador.

3.2.2 *Simulation Runtime Controller*

O processamento do simulador é controlado pelo *Simulation Runtime Controller*, que é o responsável pela inicialização de todos módulos, e pelo escalonamento das tarefas fornecidas por estes.

O comportamento dos nós na vida real, depende da interação com outros nós, dessa forma os módulos de mobilidade têm de ter em atenção

2 http://gmsf.sourceforge.net/

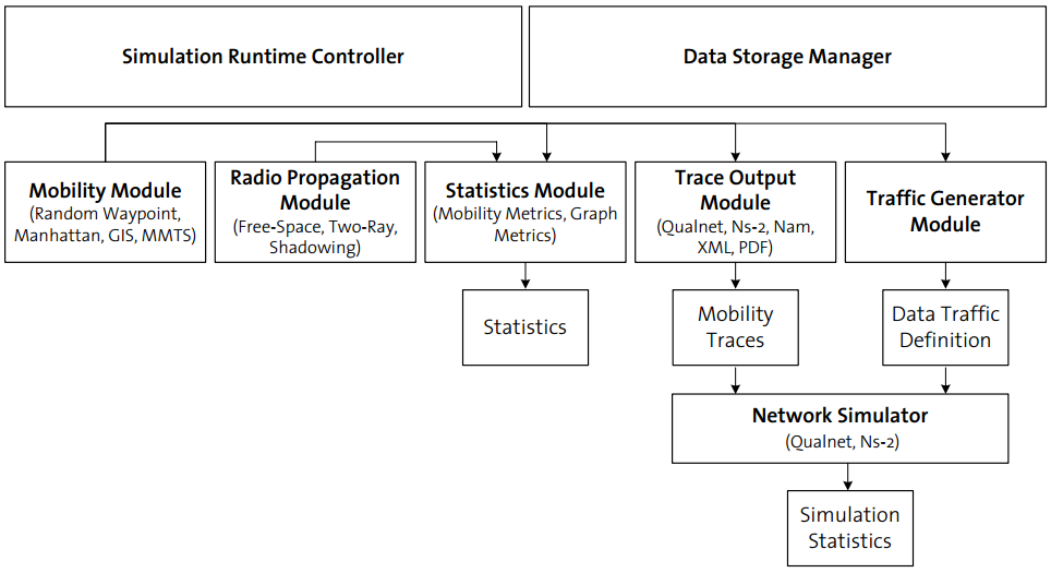


Figura 14: GMSF: Arquitetura [16]

a posição atual, velocidade e direção dos outros nós, para calcular os movimentos de cada nó, a cada instante de tempo. Para isto, foi adotada a técnica de haver uma divisão da simulação em intervalos de tempo, permitindo assim os módulos de mobilidade atualizarem as posições para cada instante de tempo de forma discreta. É então o *Simulation Runtime Controller* que divide a simulação em definidos períodos de tempo, tal como mostra a Figura 15.

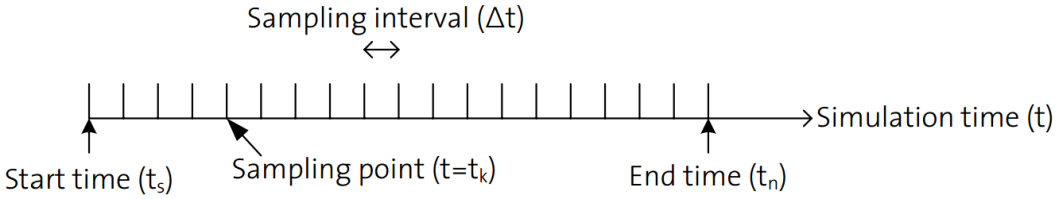


Figura 15: GMSF: Divisão Temporal [16]

No início da simulação, o *Simulation Runtime Controller* começa por inicializar o modelo de mobilidade e os módulo adicionais. A cada amostragem de tempo t_k, o tempo é atualizado e são feitas as seguintes operações:

1. É pedido ao modelo de mobilidade para atualizar as posições dos nós, de acordo com a atual amostragem de tempo.

2. É pedido aos restantes módulos para efetuarem as suas tarefas.

3.2.3 *Data Storage Manager*

O *Data Storage Manager* disponibiliza módulos que detêm acesso a dados relevantes da simulação. Com o decorrer desta, este módulo vai mantendo uma lista de todos os nós a correr na simulação, e registando ainda todos os eventos de mobilidade que aconteçam durante todo o período de simulação.

3.2.4 *Mobility Module*

Este módulo é responsável pela instanciação dinâmica da mobilidade dos nós. Cada nó tem a sua fila de eventos de mobilidade, que são gerados e inseridos pelos modelos de mobilidade. O GMSF dispõe de 6 modelos de mobilidade implementados, os quais:

- Random Waypoint model
- Manhattan model
- GIS model
- MMTS model
- Fixed node model
- Test model (for testing)

3.2.5 *Radio Propagation Module*

Este módulo tem como função determinar o alcance das comunicações entre os nós moveis. Dois nós só conseguem comunicar se um nó conseguir transmitir um sinal, e que no receptor a potência do sinal recebido seja suficiente à sensibilidade do receptor. Estes cálculos são definidos por modelos de propagação rádio, para estimar a perda de sinal como uma função da distância entre dois nós.

Cada módulo de propagação, dispõe um método chamado *isInRadioRange* usado pelos outros módulos para saber se dois nós estão em raio de comunicação. Os modelos de propagação implementados, são os seguintes:

- Free-space model
- Two-ray model
- Shadowing model
- Fixed communication range model

3.2.6 *Data Traffic Generator Modules*

Este módulo gera um ficheiro que define o tráfego de dados que deve ser transmitido entre uma rede de nós durante uma simulação. Está implementada uma classe geradora de trafego *CBRTrafficGenerator*, que gera trafego com uma taxa de dados constante entre um número específico de pares origem-destino.

3.2.7 *Traces Output Module*

Este módulo tem acesso a todos os eventos de mobilidade de uma simulação, e é responsável por exportar o histórico de posições dos nós durante a simulação. Os modelos de formatação de saída implementados, são os seguintes:

- ns-2 mobility trace format
- nam mobility trace format
- Qualnet mobility trace format
- XML mobility trace format
- PDF containing mobility traces as lines

3.2.8 *Visualization Module*

Este simulador dispõe ainda de uma interface gráfica (GUI), que mostra ao utilizador tanto as posições atuais dos nós, tal como o gráfico da rede. Poderão ser exportados para documentos PDF *screenshots* do módulo de visualização. A Figura 16 mostra o GUI do simulador, onde do lado esquerdo se podem ver as posições atuais dos nós na área de simulação, e no lado direito, a topologia de rede.

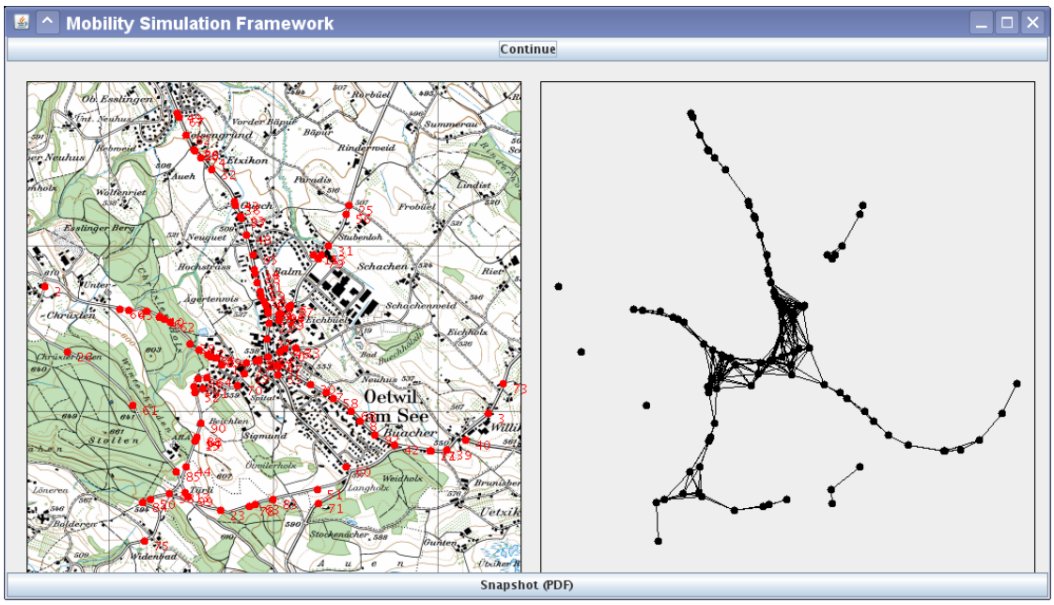


Figura 16: GMSF: GUI [16]

3.2.9 *Conclusões*

Com o estudo da documentação deste simulador, foi possível captar técnicas e ideias que poderiam ser úteis para a análise e desenvolvimento deste projeto, tal como o módulo de geração de tráfego ou os módulos de propagação rádio. Contudo embora esteja documentado, a versão do simulador disponibilizada apenas oferece os módulos de mobilidade, sendo dessa forma impossível fazer uma análise à construção dos módulos de geração de tráfego e propagação radio.

3.3 SIMULATION OF URBAN MOBILITY

O Simulation of Urban MObility (SUMO) nasceu em 2000 e é uma aplicação open-source que começou a ser desenvolvida em C++ pelos funcionários do Centro Aéreo-Espacial Alemão (DLR), com o fim de criar uma solução para dar apoio à comunidade investigadora de tráfego rodoviário, fazendo simulações, mas possibilitando também ser usado como uma ferramenta de suporte para implementar e avaliar os próprios algoritmos de mobilidade. Este software encontra-se atualmente na versão 0.17.1 lançada em Maio de 2013, e pode ser descarregada no site online[3] do projeto.

3 http://sumo.sourceforge.net

3.3.1 *Mapas*

Este simulador dispõe de dois mecanismos para fazer o carregamento das redes de caminhos (mapas), por onde os nós se deslocam durante a simulação.

A primeira forma é fazer o carregamento de mapas existentes, podendo estes ser em diversos formatos distintos, os quais:

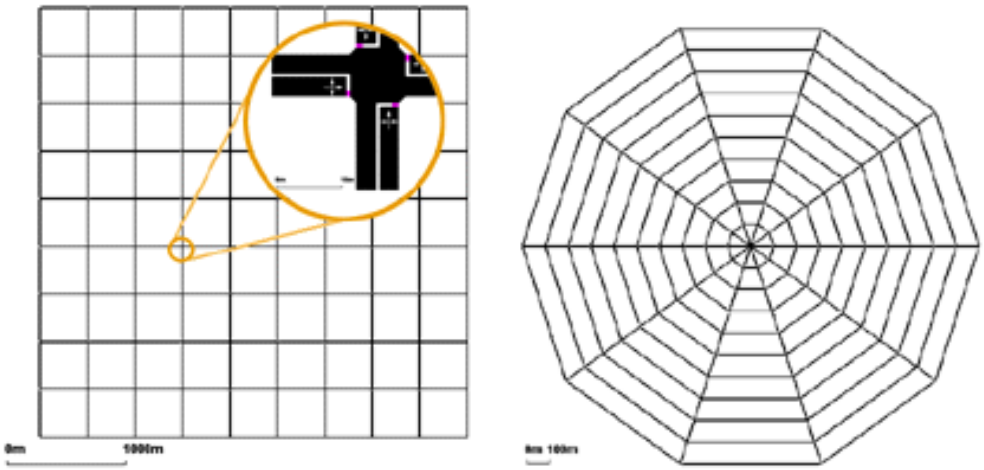
- OpenStreetMap
- PTV VISUM
- PTV VISSIM
- openDRIVE
- MATsim
- ArcView-data base
- Elmar Brockfelds unsplitted and splitted NavTeq-data
- RoboCup Rescue League

Qualquer um destes tipos de mapa podem ser carregados usando o módulo *NETCONVERT*, que converte o mapa para o formato SUMO.

O outro método que este simulador dispõe para fazer o carregamento de mapas, consiste na geração dos mesmos. O módulo *NETGENERATE* permite que sejam gerados mapas no formato SUMO para serem usados na simulação, existindo três métodos distintos para a geração dos mapas: *Grid-like*, *Spider-like* e *Random*. A Figura 17 mostra mapas exemplo, para cada tipo de geração de mapas.

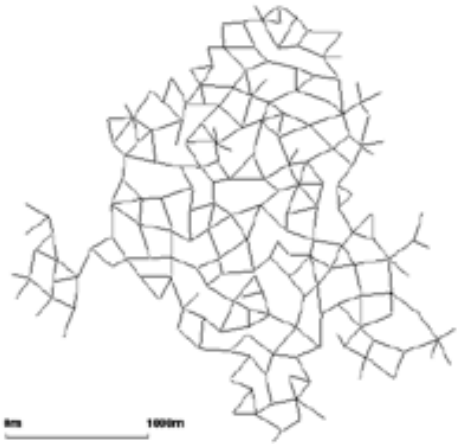
3.3.2 *Mobilidade*

Para realizar uma simulação, é necessário à priori definir quais os agentes que farão parte desta, e qual o comportamento que terão durante a simulação. Para tal, para correr a simulação, o simulador necessita saber quais as rotas que um nó irá percorrer durante esta. Existe assim um conjunto de diferentes técnicas para geração de rotas, as quais:



(a) Grid-like

(b) Spider-like



(c) Random

Figura 17: SUMO: Diferentes tipos de geração de Mapas [12]

- *Trip*

  Uma *trip* define uma viagem, com um ponto de origem, um destino e um instante de tempo em que começa a viagem. Assim com o módulo *DUAROUTER* podem-se converter as viagens em rotas.

- *Flow*

  Este método usa a mesma lógica do método anterior, contudo permite atribuir o mesmo caminho para vários veículos.

- *Random*

  Com o uso deste método, é possível gerar facilmente rotas de forma aleatória.

- *OD-Matrices*

Este método usa matrizes origem-destino para gerar as rotas. Estas matrizes são normalmente usadas por entidades reguladores de tráfego.

- *Detector data (observation points)*

  Este método utiliza dados fornecidos por autoridades reguladores de tráfego, que identificam padrões de circulação, permitindo assim induzir ciclos ou outros métodos para o controlo de tráfego.

- *By hand*

  Tal como o nome indica, é também possível gerar os ficheiros de rotas de forma manual.

- *Population statistics*

  Este método traduz dados estatísticos de população na geração de trafego, permitindo assim tornar a simulação mais realista gerando mais ou menos tráfego nas zonas com mais ou menos população.

Este simulador permite também controlar um conjunto de parâmetros que definem um nó, permitindo desta forma simular vários tipos de veículos ou ate pessoas. Alguns dos parâmetros configuráveis são:

- Aceleração
- Velocidade de reação
- Comprimento do veículo
- Velocidade máxima

É também possível definir paragens com determinada duração para um determinado veículo, permitindo assim por exemplo, que durante uma rota na simulação, este faça as paragens definidas de determinado transporte público.

Outra possibilidade na simulação de mobilidade, é a definição de semáforos, sendo possível definir a sequência e durações das cores do semáforo.

3.3.3 *Resultados*

Como saída, este simulador permite exportar vários dados relativos a uma simulação, tais como:

- Posições dos veículos para cada instante de tempo (no formato ns-2)
- Posições de determinado tipo de veículos para cada instante de tempo
- Nível de poluição calculada para um determinado ponto
- Nível de poluição acústica calculada para um determinado ponto
- Informação das rotas tomadas por um determinado veículo durante a simulação

3.3.4 *Visualização*

Para além da exportação dos resultados da simulação, é também possível ver a simulação numa interface gráfica animada. A Figura 18 mostra esta interface.

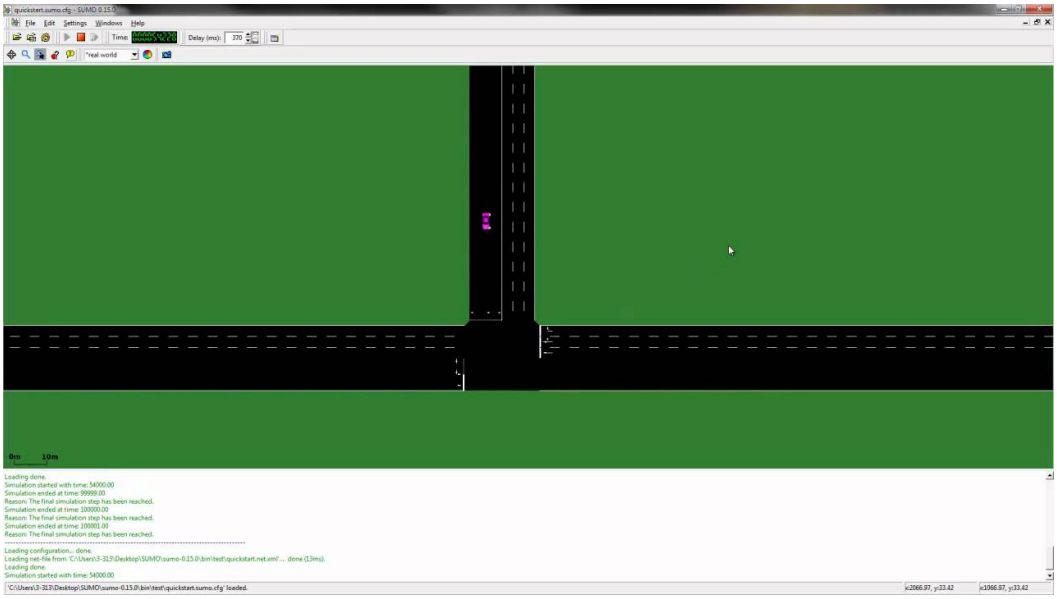


Figura 18: SUMO: GUI

3.3.5 *Conclusões*

Como esta dissertação se foca na simulação das comunicações entre nós e na simulação da tecnologia *Bluetooth*, não será tido em conta este simula-

dor como fonte de inspiração, pois este não dispõe de mecanismo de troca de mensagens, focando-se apenas na mobilidade.

3.4 THE ONE

O simulador The One (Oportunistic Networking Environment)[8] foi desenhado e desenvolvido com o intuito de estudar e avaliar as técnicas de "DTN routing" tal como protocolos para a sua aplicação. Desenvolvido na linguagem de programação Java, o simulador The One permite assim aos utilizadores criar cenários baseados em diferentes modelos de movimento. Disponibiliza 6 protocolos de encaminhamento bem conhecidos, e oferece ainda um ambiente para que possam ser implementados outros. Dispõe ainda de um interface gráfico interativo tal como mostra a Figura 19, onde é permitido ao utilizador ver a disposição dos nós, movimentação e comunicação entre eles.

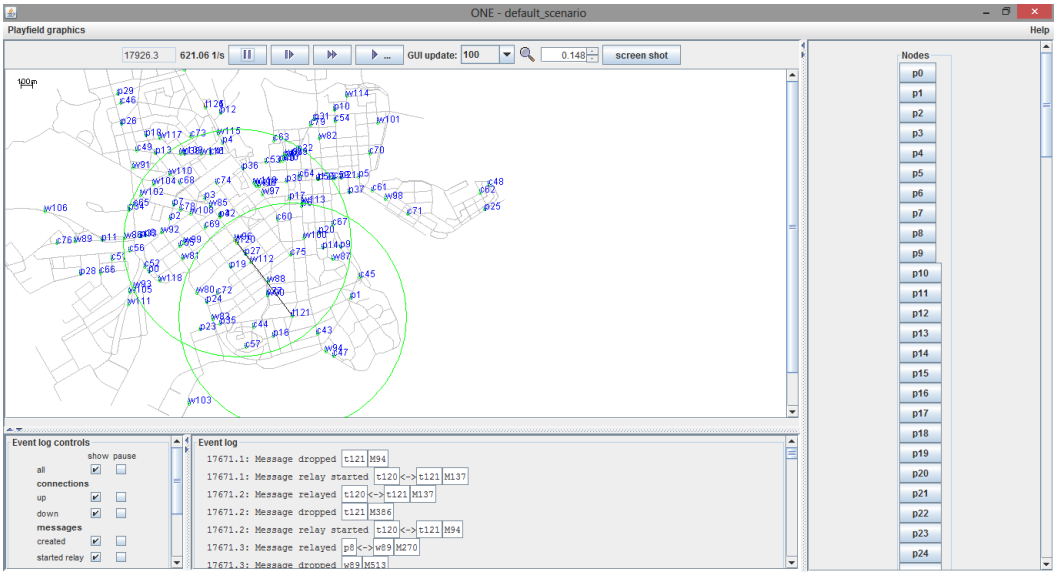


Figura 19: The One: Interface Gráfico

Neste projeto, a análise e estudo do simulador The One teve uma importância significativa pois este simulador possui soluções de comunicações oportunistas, as quais foram tidas em conta para o estudo e análise de possíveis soluções para o nosso projeto. É de todo importante conhecer a forma como o simulador The One trata das comunicações entre nós, e poder ter então em conta essas características para construir a melhor solução possível neste projeto.

3.4.1 *Arquitetura*

Após ter sido feito um estudo minucioso ao simulador The One foi possível entender a sua arquitetura (Figura 20) e dessa forma perceber como este foi desenvolvido.

As principais funções do simulador The One são a modelação do movimento dos nós, a comunicação entre nós, a gestão das mensagens geradas e recebidas e respetivo encaminhamento destas. O conjunto de resultados e análises são ainda gerados e disponibilizados através de um interface de visualização gráfico, relatórios e ferramentas de pós-processamento.

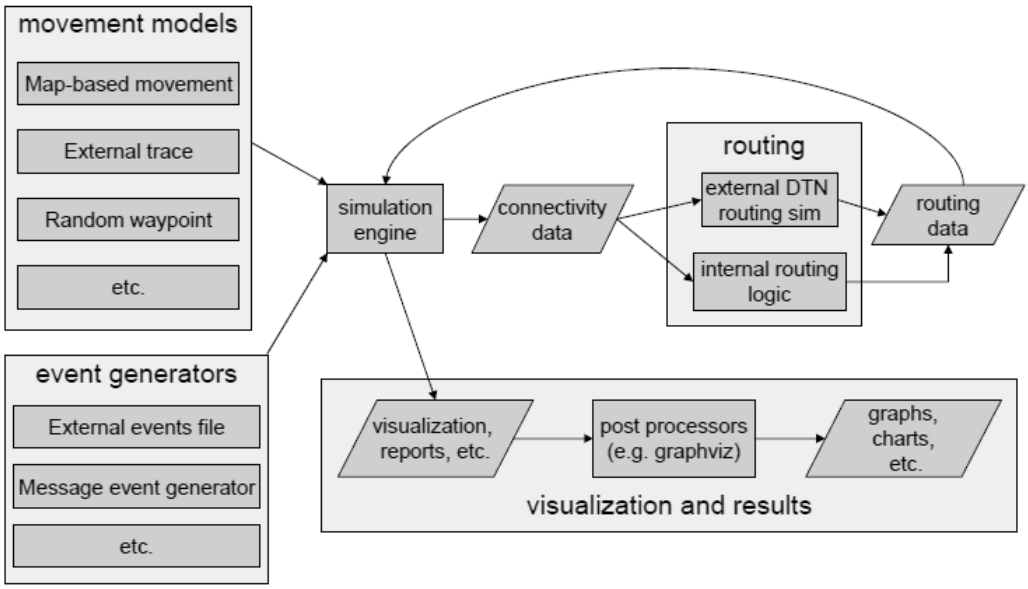


Figura 20: The One: Arquitetura [9]

3.4.2 *Nós*

No simulador, um nó representa um ator na simulação, e os nós estão organizados por grupos. Cada grupo partilha um mesmo conjunto de parâmetros tais como o tamanho do buffer de mensagens, o alcance de rádio e o modelo de mobilidade. Desta forma, visto que diferentes grupos podem ter diferentes configurações, torna-se possível usar os nós e diferenciá-los como pedestres, carros, transportes públicos, etc.

3.4.3 *Mobilidade*

Os modelos de mobilidade definem como se irão mover os nós durante a simulação. Foram então implementados modelos de mobilidade como por exemplo movimento baseado num mapa ou movimento baseado em *waypoints* aleatórios. É ainda possível importar dados de mobilidade de fontes externas.

3.4.4 *Encaminhamento*

Enquanto os modelos de mobilidade definem para onde os nós se devem mover, os módulos de encaminhamento definem o que fazer às mensagens geradas. O The One tem implementados 6 algoritmos de encaminhamento bem conhecidos que são: *First Contact*, *Direct Delivery*, *Spray and Wait*, *Epidemic*, *PRoPHET* and MaxProp. Conta ainda com um módulo de encaminhamento passivo que pode ser usado para interagir com simuladores de routing DTN externos. Quando 2 ou mais nós se encontram suficientemente perto entre si (tendo em conta as características de alcance) cria-se a oportunidade de estes trocarem mensagens. Qualquer modelo de encaminhamento tem como primeira tarefa ver se algum nó tem mensagens destinadas a outros nós e procura enviá-las. Se a mensagem já foi anteriormente entregue, o nó descarta-a.

O algoritmo de encaminhamento *Direct Delivery*, não transmite qualquer mensagem, a não ser que esteja em contacto direto com o destinatário. Este método poupa espaço no buffer e largura de banda, no entanto não é obviamente uma solução ótima para a maioria dos casos, pois a probabilidade de uma mensagem ser entregue é reduzida.

O algoritmo de encaminhamento *Epidemic* usa uma abordagem diferente. Quando está suficientemente próximo de um outro nó, para além de lhe enviar as mensagens em que ele é o destinatário, envia também todas as outras mensagens que possui, até que se perca a conexão, isto para alargar a probabilidade da entrega da mensagem ao máximo. Nesta situação caso se tenha largura de banda ilimitada ou buffer ilimitado, o resultado será a máxima disseminação das mensagens.

O algoritmo *First Contact* envia o máximo de mensagens para os outros nós enquanto tem uma conexão estabelecida e remove a cópia local após a

mensagem ter sido enviada com sucesso. Cada nó apenas aceita a mensagem, se esta ainda não tiver passado por si. Este algoritmo faz assim, com que haja apenas uma única cópia da mensagem na rede, mas não garante que o primeiro nó a quem enviou a mensagem seria melhor candidato que o nó anterior.

O algoritmo *Spray and Wait*, funciona de forma parecida como o *Epidemic*, no entanto é um pouco mais complexo, pois condiciona o número de cópias da mensagem na rede. A origem faz N réplicas para os primeiros N contactos, e estes só entregam no destino.

Por fim os algoritmos *PRoPHET* e *MaxProp* são bastante mais complexos, pois estes guardam o trajeto que foi feito pelas mensagens nas passagens pelos nós. Esta informação pode então ser usada no caso de um nó ser um bom candidato a estar conectado com o destinatário da mensagem, assumindo que estes já estiveram conectados anteriormente. A diferença entre estes 2 algoritmos é que o *PRoPHETE* verifica se um nó é mais provável que outro a receber a mensagem, enquanto que o MaxProp usa o algoritmo de *Dijkstra* para calcular os caminhos de nó a nó usando o conhecimento das probabilidades.

3.4.5 *Implementação*

O simulador The One foi desenvolvido na linguagem de programação Java, linguagem esta orientada a objetos. Partes diferentes do programa estão dividias em pacotes diferentes, tal como mostra a Figura 21, onde são mostradas também as dependências entre pacotes.

A parte *core* do simulador contém as classes que representam um host ou uma conexão. As classes relacionadas com a interface gráfica estão no pacote GUI, que contém também um sub-pacote *playfield*, contendo este as classes relacionadas com os objetos gráficos para a visualização dos mapas. A classe generic user interface e a classe text-based console, estão inseridas no pacote UI. Este pacote tal como o pacote GUI iniciam as classes *SimScenario* e a classe *World* a partir do core, que por sua vez cria então as instâncias dos módulos de encaminhamento do pacote de encaminhamento, e as instâncias dos módulos de movimento do pacote de movimento. Durante a simulação os módulos de movimento e de encaminhamento, geram dados para os módulos do pacote *report*. Por fim o pacote *test*, que não está diretamente relacionado com o simulador, contém um conjunto de testes

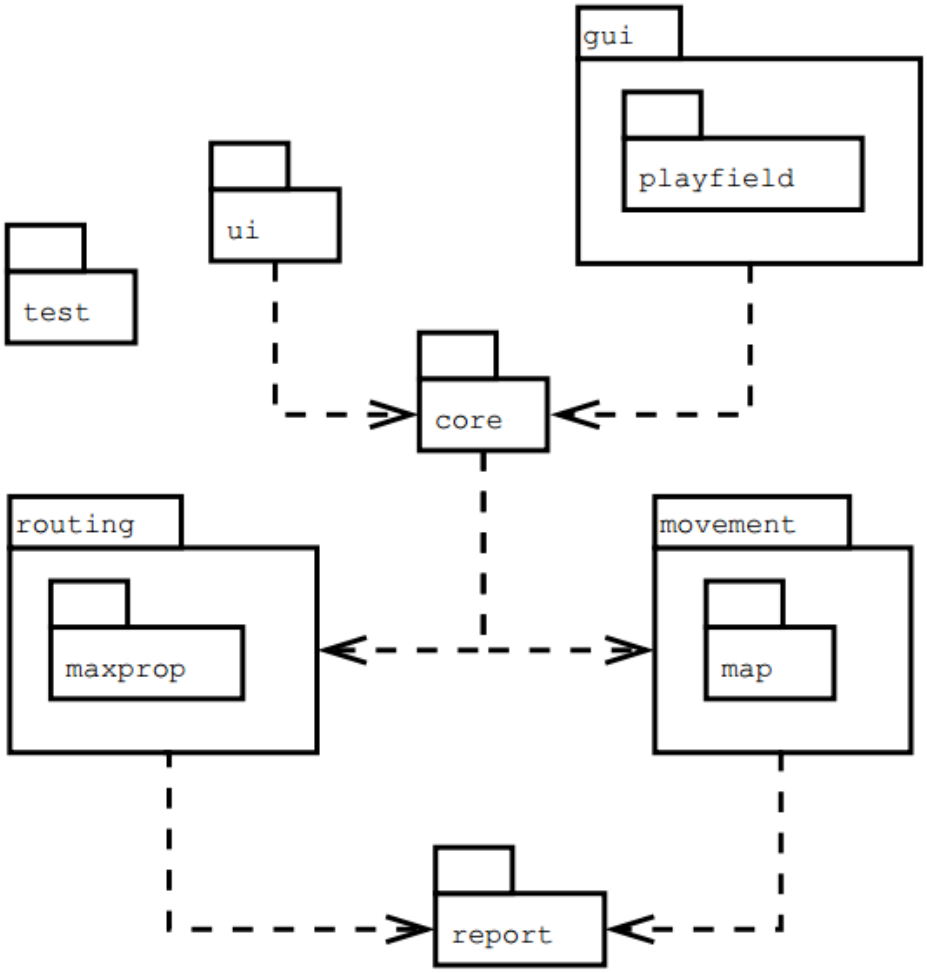


Figura 21: The One: Software Packages[8]

ao sistema e às unidades, que podem ser usados para verificar se tudo corre como esperado.

3.4.5.1 *Core*

A parte core da aplicação é onde estão presentes as classes mais importantes, tais como as classes que definem um nó, uma conexão, uma mensagem, um interface, etc.

A classe *DTNHost* define um nó, tem assim nela representadas todas as variáveis que a definem, tal como o seu nome, a sua posição, os seus interfaces, o seu router, o seu modelo de movimento, etc.

A classe *NetworkInterface*, é a classe que trata das conexões entre nós, e é onde estão definidas as características de um tipo de conexão, tal como o alcance de transmissão, a velocidade de transmissão ou o intervalo de

tempo entre leituras. Esta classe tem ainda implementados os métodos para estabelecer ou terminar uma conexão, verificação de alcance para transmissão, etc. Cada nó tem assim uma lista de interfaces em representação das possíveis tecnologias que este poderá ter, tal como bluetooth, 802.11.g, etc.

Cada objeto *NetworkInterface* contém uma lista de ligações entre nós, sendo cada ligação definida por um objeto *Connection*. Este objeto por sua vez, é o objeto que define uma conexão entre dois nós. É também nesta classe que estão implementados os métodos de gestão de transferência de mensagens, tais como começar uma transferência, abortar uma transferência, finalizar uma transferência, etc.

3.4.5.2 *Módulos de Movimento*

A Figura 22 mostra as classes mais importantes do pacote *movement*. Tal como se pode observar, todos os módulos de movimentação derivam da classe *MovementModel*, que disponibiliza a interface aos pedidos de um novo caminho para os nós. Dessa forma as sub-classes disponibilizam diferentes soluções de possibilidades de movimento.

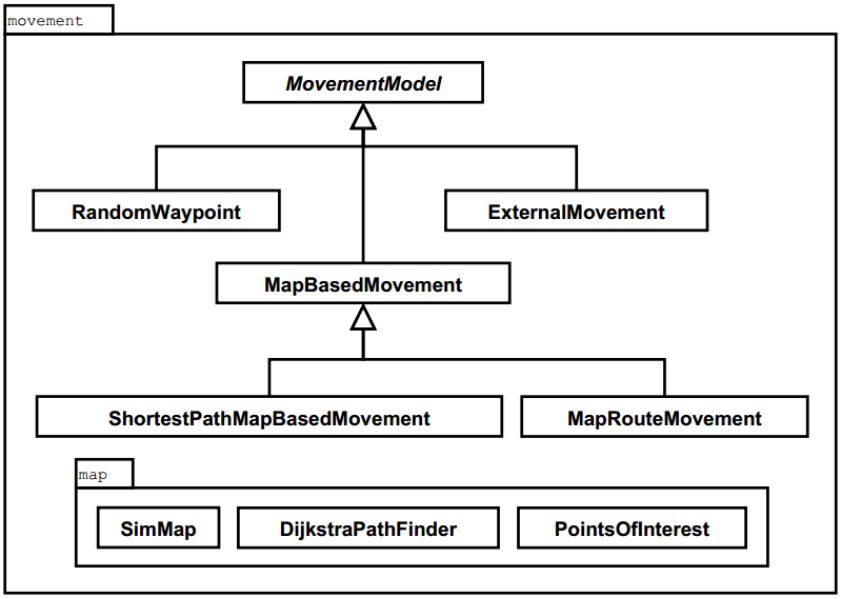


Figura 22: The One: Movement Package[8]

É ainda possível ver na Figura 22 um sub-pacote map, que dispõe de classes utilitárias aos modelos *MapBasedMovement*. É exemplo disso a classe *SimMap* que dispõe os dados do mapa, a classe *DijkstraPathFinder*

que é usada para achar o caminho mais curto entre dois nós usando o algoritmo de *Dijkstra* com a informação do mapa, ou ainda a classe *PointsOfInterest* que se preocupa com a leitura de dados sobre os pontos de interesse e selecionando os apropriados de acordo com as configurações definidas.

3.4.5.3 *Módulos de Encaminhamento*

Com uma estrutura parecida ao pacote de movimento, tal como mostra a Figura 23, o pacote de encaminhamento é composto por uma classe *MessageRouter* que dispõe de uma interface e de funcionalidades para os módulos de encaminhamento que derivam assim desta classe. Desta forma é a classe *MessageRouter* que tem a responsabilidade de armazenar a informação das mensagens de que o nó tem atualmente no seu buffer, informação das mensagens que este está atualmente a receber nas suas conexões ativas e ainda a informação das mensagens já recebidas como destinatário final que já foram encaminhadas para a camada de aplicação, e portanto já não se encontram assim presentes no buffer.

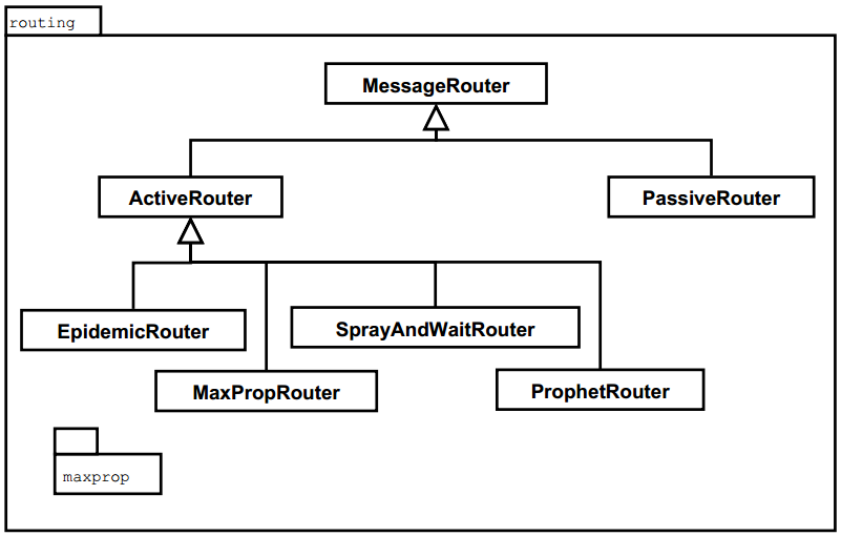


Figura 23: The One: Rooting Package[8]

A classe *MessageRouter* aceita qualquer mensagem, porém os módulos de encaminhamento poderão aceitar ou não dependendo da sua lógica de gestão. Com o método de retorno, os módulos de encaminhamento poderão avisar qualquer que seja o nó, para que estes enviem a mensagem mais tarde, ou simplesmente parem de a enviar. Assim sendo, quando um nó pretende transferir uma mensagem para outro nó, este pede ao

devido objeto conexão para iniciar uma transferência, onde este por sua vez encaminha o pedido para o outro nó. O outro nó chama o método *receiveMessage* do seu módulo de encaminhamento, e então o módulo de encaminhamento pode verificar se quer ou não receber a mensagem. Este poderá portanto aceitar ou rejeitar a mensagem, por exemplo no caso de esta já se encontrar no seu buffer.

O método de recepção de mensagens dispõe ainda de métodos que podem ser chamados quando uma mensagem foi transferida com sucesso, quando uma mensagem foi abortada, quando uma nova mensagem deverá ser gerada ou ainda quando uma mensagem deverá ser apagada do buffer. Estes métodos são usados para informar detetores de inventos, tal como módulos de *report* ou eventos do modo gráfico.

A sub-classe *ActiveRouter* dispõe da implementação de dois importantes métodos: *changedConnection* e *update*. O primeiro é chamado sempre que surge uma nova conexão ou alguma é terminada, enquanto que o segundo é usado para cada atualização. A classe *ActiveRouter* fornece ainda tarefas comuns a todos os módulos de encaminhamento, tais como o envio de mensagens, ou a tentativa do envio de um conjunto de mensagens numa específica ordem, usando um conjunto de conexões recebidas. Desta forma um simples modo de encaminhamento poderá ser construído apenas com algumas linhas de código, visto que não necessitará de muitas mais funções.

3.4.6 *Conclusões*

Com a análise ao simulador The One, foi possível observar que este simula de forma muito genérica o protocolo de comunicação, possibilitando apenas modificar as constantes velocidade e raio de transmissão, para cada interface. É também de notar que as mensagens enviadas funcionam também de forma bastante genérica não respeitando qualquer protocolo de comunicação como o Bluetooth ou 802.11.g. O simulador The One não implementa funcionalidades ao nível da camada física ou da camada de ligação, não será assim tido em conta como fonte de inspiração no que respeita ao protocolo de comunicação nestes níveis da camada. Contudo, foi possível perceber como é feita toda a gestão das mensagens. Cada nó tem uma lista de interfaces, onde cada uma representa um dispositivo de comunicação, onde é constante de cada interface o raio de transmissão

ou a velocidade de transmissão. Cada interface tem uma lista de conexões, onde se encontram todas as conexões estabelecidas no momento entre esse e outro qualquer nó. Dessa forma, através de um objeto router cada nó envia então as suas mensagens para qualquer outro nó.

Embora não possam ter sidos em conta aspectos deste simulador no que respeita às camadas física e de ligação, foram então tidas em conta estas características do simulador The One, no que respeita à forma de gestão e de envio de mensagens.

3.5 VISSIM

Desenvolvido pela empresa alemã Planung Transport Verkehr AG (PTV), o Verkehr In Städten - SIMulationsmodell (VISSIM) é um software de simulação de mobilidade, que começou a ser desenvolvido em 1992 e é hoje líder mundial no mercado. Desenvolvido em C++, o VISSIM é um software pago de código fechado e especificamente desenvolvido para fazer a simulação de modelos microscópicos de tráfego, incluindo transportes públicos e mobilidade pedestre, simulando assim o fluxo de tráfego multimodal[4], de carros, veículos de mercadorias, autocarros, comboios, trams, metros, motociclos, bicicletas e pessoas.

O PTV VISSIM é uma parte do PTV Vision Traffic Suite, que inclui também o PTV Visum que permite fazer análises e previsões de tráfego, e o PTV Vistro, que tem como fim fazer otimizações nas sinalizações rodoviárias, avaliar impactos das modificações, e consegue ainda gerir múltiplos cenários, gerando informação da análise do tráfego. O software encontra-se atualmente na versão 5.40 e poderá ser feito o download de uma versão demostrativa na página online[5] do simulador.

3.5.1 *Arquitetura*

Qualquer simulador de tráfego, necessita de um modelo matemático para representar o sistema de movimentação, para simular os aspetos técnicos e organizacionais do movimento. É também necessário um módulo para gerir os veículos e pessoas na simulação, e visto ser este um simulador

4 A simulação de tráfego multimodal, descreve a habilidade de simular diferentes tipos de tráfego.

5 http://vision-traffic.ptvgroup.com/en-us/products/ptv-vissim/

com modelos microscópicos de tráfego, existe também um módulo para fazer o controlo do tráfego que tem de ser bastante detalhado. Estes três módulos são dependentes uns dos outros, tal como as setas da Figura 24 mostram, sendo que durante uma simulação, os três módulos encontram-se ativos e com constantes dependências ativas de outros módulos. Para além destes três módulos, existe ainda um outro que é responsável pela saída de resultados das simulações.

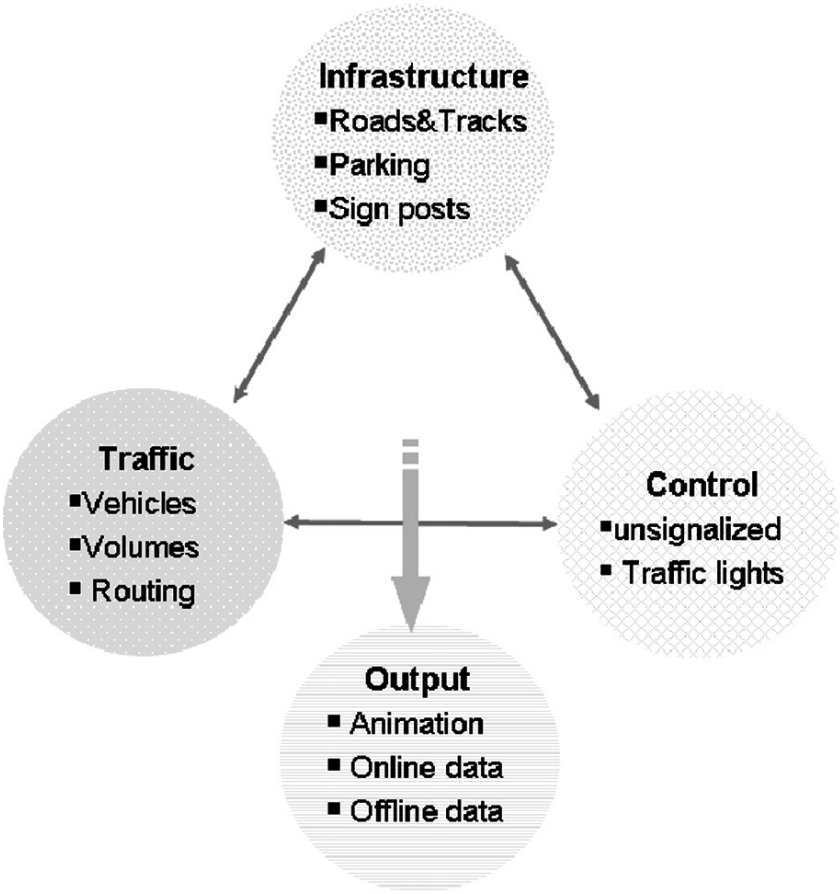


Figura 24: VISSIM: Arquitetura [10]

3.5.2 *Infraestrutura*

Este módulo que representa a infraestrutura de estradas e caminhos de ferro, inclui também sinais de trânsito e lugares de estacionamento, sendo então o bloco fundamental necessário para modelar as estradas e caminhos físicos. As paragens de transportes públicos e os lugares de estacionamento são necessários para determinar o começo e fim das viagens.

São elementos da infraestrutura:

- Ligações e Conetores
  Uma ligação representa um caminho e um conetor uma junção de dois ou mais caminhos
- Sinais de limite de velocidade
- Sinais de stop e cedência de passagem
- Semáforos
- Lugares de Estacionamento

3.5.3 *Tráfego*

Este módulo define os aspetos técnicos de um veículo e as especificações dos fluxos de tráfego, que é definido ou por matrizes origem-destino, ou por geradores de tráfego. O movimento de transportes públicos é também tarefa deste módulo que define a sequência das ligações e paragens.

Existem dois tipos de tráfego, o privado e o público. Enquanto que os veículos de transporte privado determinam as suas rotas individuais, os veículos de transporte público irão seguir de forma predeterminada, caminhos com paragens. Autocarros e serviços não regulares, são tratados como tráfego privado.

3.5.3.1 *Tráfego Privado*

A classe do tráfego privado está classificada em categorias como camiões, carros, bicicletas ou pedestres. Para cada categoria representante de um tipo de veículo, estão especificadas diferentes características relativas a esse veículo tais como:

- Comprimento
- Largura
- Aceleração
- Desaceleração
- Velocidade Máxima

Dependendo no objetivo da simulação os dados relativos aos veículos poderão ser simplificados distribuindo a especificação destes diferentes tipos de veículos. Os veículos são gerados de forma aleatória, em estradas ou em lugares de estacionamento.

3.5.3.2 *Tráfego Público*

Todas as características dos transportes privados, são também relevantes para os transportes púbicos, contudo são necessárias ainda mais características para definição dos tempos de paragem. Uma linha de transportes públicos, consiste em autocarros, trams, ou metros a seguirem uma sequência de paragens definidas de acordo com um determinado tempo.

3.5.4 *Controlo*

Este módulo contém todos os elementos necessários ao controlo do tráfego. Todas regras de circulação tais como regras em cruzamentos, prioridade da direita ou resposta a sinais de transito, são definidas neste bloco. Embora a sinalização rodoviária, seja um elemento do módulo de infraestrutura, as definições referentes ao comportamento à sinalização, pertencem ao módulo de controlo.

3.5.4.1 *Interseções não sinalizadas*

Na movimentação, quando existe ou não direito de passagem na falta de sinalização, esta é modelada com regras de prioridade. Isto é aplicado em todas as situações onde veículos em diferentes linhas ou conetores de linhas, se devem reconhecer para definir prioridades de passagem.

As regras de prioridade são usadas nas seguintes condições:

- Interseções onde o tráfego tem de ceder passagem à direita
- Interseções onde o tráfego na rua que termina, tem de ceder passagem ao tráfego da rua que se perlonga
- Interseções em ruas de duplo sentido ou sentido único, controladas por stop
- Rotundas, onde os veículos que entram têm de ceder passagem aos que já se encontram na rotunda

- Vias de aceleração, onde o tráfego que entra tem de ceder passagem ao que já se encontrava nessa via
- Cedência de passagem a autocarros, se estes estiverem a sair de uma paragem e com sinalização ativa

3.5.4.2 *Interseções sinalizadas*

Embora os semáforos sejam parte da infraestrutura, o controlo destes é uma função do módulo de controlo. Vários semáforos conetados entre si pertencem a um mesmo grupo, onde cada grupo detém uma unidade controladora.

É possível definir também controladores de tráfego, que dão informações relativas à quantidade de tráfego a cada instante, possibilitando assim estimar o atraso e comprimento de filas, permitindo que com a unidade controladora de um grupo de semáforos, seja possível controlar a cada instante o tempo de cada semáforo individual.

O VISSIM disponibiliza ainda uma API estruturada tal como uma linguagem de programação *C* ou *Pascal*, acompanhada de algumas funções relevantes para a engenharia de tráfego e ainda de um editor gráfico para diagramas de fluxo.

3.5.5 *Resultados*

Este módulo é responsável por todos os tipos de resultados de saída das simulações, processando assim dados provenientes dos três módulos, sem que lhes dê qualquer resposta. Os resultados tanto poderão ser gerados durante a simulação, tal como mostrados com representações animadas de veículos e estados de controlo de tráfego, ou ainda como dados estatísticos e estados de veículos apresentados numa caixa de texto.

A movimentação de todos os veículos pode ser graficamente animada a duas ou três dimensões, permitindo ainda gerar vídeos em formato *.avi* das simulações. Para melhor visualização, poderão ainda ser definidos fundos com fotos aéreas, modelos CAD[6], ou ainda serem importados modelos Google Sketchup[7]. A Figura 25 mostra a interface gráfica 3D de uma

6 CAD, do inglês: computer aided design. Desenho assistido por computador
7 O Sketchup é um software da Google, usado para criar modelos 3D

simulação.

Figura 25: VISSIM: GUI

É também possível exportar dados relativos da simulação, tais como atrasos, tempos de viagens, paragens, filas, velocidades, etc. Esta informação pode ser sintetizada para toda a simulação ou em blocos de tempo definidos pelo utilizador. É possível guardar os dados em formato texto, ou o VISSIM oferece ainda a possibilidade de exportar os dados já formatados para serem importados pelo Microsoft Excel ou Access.

3.5.6 *Conclusões*

Como esta dissertação se foca na simulação das comunicações entre nós e na simulação da tecnologia *Bluetooth*, não será tido em conta este simulador como fonte de inspiração, pois este não dispões de mecanismo de troca de mensagens, focando-se apenas na mobilidade.

4

BARTOLOMEU URBAN MOBILITY SIMULATOR

Bartolomeu Dias foi um navegador português, que se tornou célebre em 1488 por atingir o feito de ser o primeiro europeu a navegar para além do extremo sul de África, dobrando assim o Cabo da Boa Esperança e chegando ao oceano Índico, através do Atlântico. Este histórico ilustre marinheiro Português, serviu assim de inspiração para o nome deste simulador, Bartolomeu Urban Mobility Simulator (BartUM).

Este simulador nasce em 2010, tendo como objetivo criar um simulador de mobilidade em ambientes urbanos em grande escala, e simular redes moveis inseridas no mesmo ambiente. Dessa forma começou a ser desenvolvido pelo Francisco Silva, como tema da sua dissertação de mestrado [15], onde implementou o *Core* deste simulador.

No ano seguinte, com vista a continuar o trabalho já realizado pelo Francisco Silva, surgiram mais dois temas para dissertação de mestrado com trabalhos neste simulador. O Rui Pinheiro desenvolveu assim a sua dissertação [14], desenvolvendo os modelos de mobilidade dos atores carro, tram e pedestre. O Laurent Miranda desenvolveu também a sua dissertação [11], implementado uma aplicação gráfica de monotorização e visualização das simulações em tempo real.

Este capitulo descreve assim todo o trabalho efetuado anteriormente no simulador, explicando a arquitetura deste e o funcionamento dos diversos componentes que o constituem. Serão ainda explicadas algumas modificações feitas no simulador com o fim de corrigir e melhorar algumas falhas que nele existiam.

4.1 ARQUITETURA

Sendo um objetivo inicial deste simulador permitir a simulação de um grande número de atores, este foi pensado e desenhado de forma distribuída, com vista a dispersar o processamento por diversas máquinas. A

arquitetura geral do simulador está assim dividida em três elementos: *Global Coordinator*, *Local Coordinator* e *Vizualization*, onde cada um destes elementos são executados separadamente, podendo ser executados na mesma máquina ou em máquinas diferentes, com o fim de distribuir o processamento. Para tal, estes terão de estar ligados entre si, através de uma rede de área local (LAN), permitindo assim a troca de informação entre eles. A Figura 26 mostra a arquitetura do simulador.

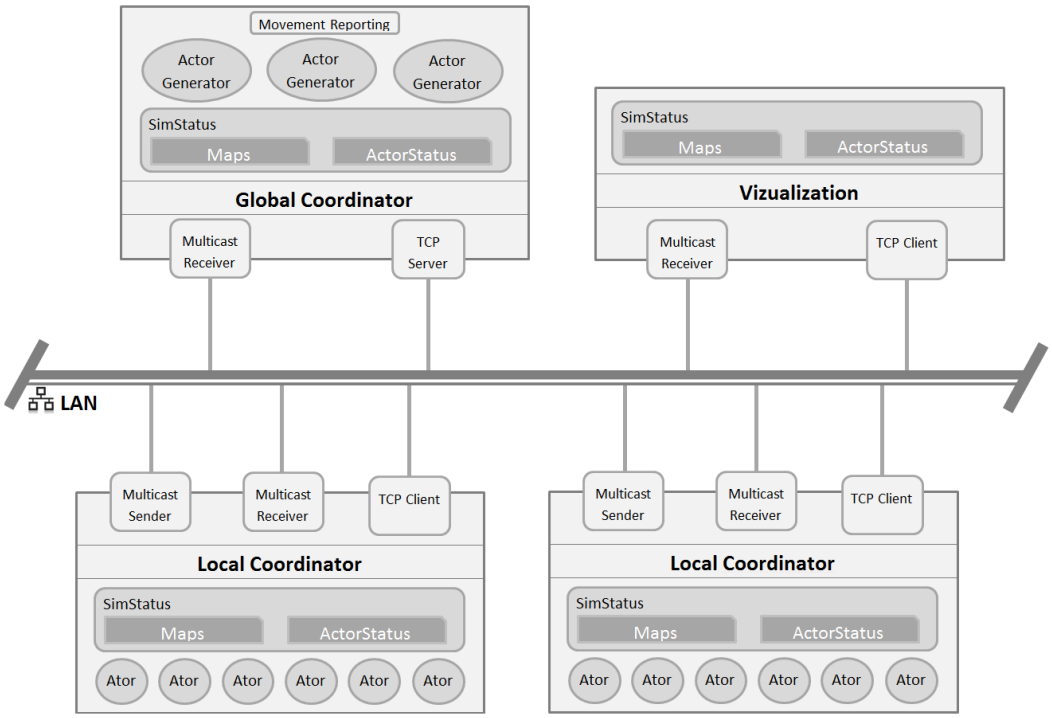


Figura 26: BartUM: Arquitetura

4.1.1 *Global Coordinator*

O *Global Coordinator* é o elemento central do simulador, encarregado de inicializar e controlar o sistema, entre outras funções. Dessa forma, apenas existe um *Global Coordinator* por simulação. Esta entidade é responsável pela configuração da simulação, lendo para isso um ficheiro definido pelo utilizador, e com ele tratando assim de inicializar a simulação, fazendo o carregamento dos mapas e criando os pretendidos geradores de tráfego, decidindo qual o elemento *Local Coordinator* que cuidará de cada ator gerado, controlando e distribuindo assim, a carga pelos diferentes *Lo-*

cal Coordinators.

4.1.2 *Local Coordinator*

O *Local Coordinator* trata-se de um coordenador local, responsável principalmente pelo controlo dos seu atores, tratando das movimentações e comunicações destes. Visto haver a procura de distribuição de carga de processamento da simulação, poderão existir diversos *Local Coordinators* numa simulação.

4.1.3 *Vizualization*

O *Viualization* permite fazer a visualização gráfica de uma simulação, permitindo desta forma ver a movimentação dos diversos atores, seja em tempo real ou ver simulações anteriormente efetuadas. Este não tem qualquer interferência com o processamento da simulação, podendo esta decorrer sem que seja executado.

Torna-se assim possível ver a simulação de forma animada com o uso dos mapas usados nesta, e sobre eles, a movimentação dos atores, permitindo que seja feito Zoom-In, Zoom-Out ou até arrastar o mapa, em busca de certos locais. A Figura 27 mostra o interface gráfico gerado pelo *Vizualization*.

4.1.4 *SimStatus*

O *SimStatus* trata-se de uma entidade responsável por armazenar dados relativos à simulação, quer sejam dados necessários para a criação desta, ou dados gerados durante.

Estando presente nos três elementos do simulador, este guarda a informação relativa aos mapas carregados para a simulação, as posições atuais dos atores locais, e também uma lista de mensagens a enviar e receber de e para outros atores locais ou não locais.

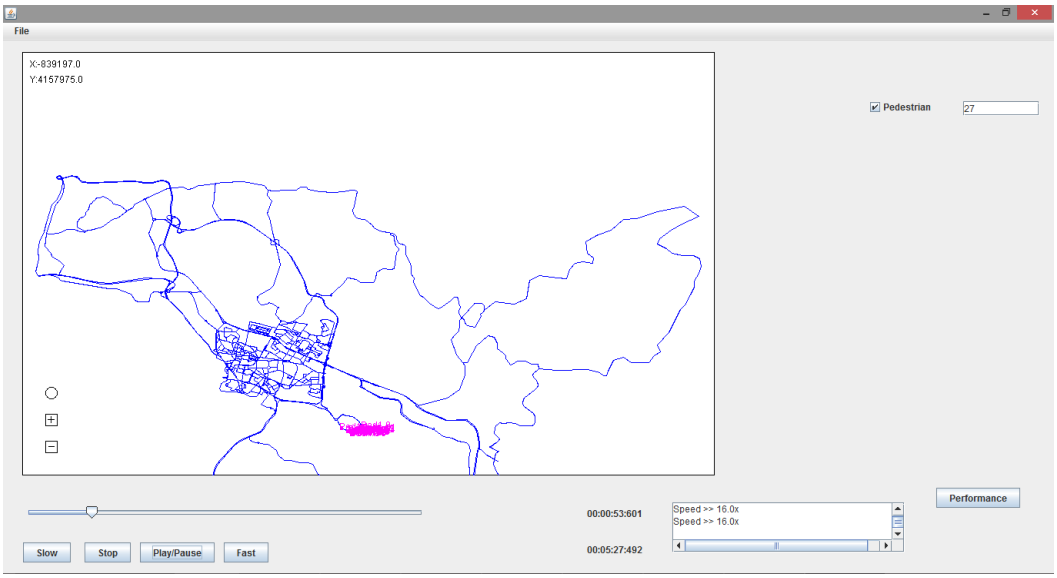


Figura 27: BartUM: Vizualization

4.1.5 *Mapas*

Com o intuito de permitir a simulação em ambientes realistas, foi estudada e desenhada uma solução para permitir o carregamento de mapas baseados em espaços reais, podendo representar assim centros urbanos tal como Tokio ou Lisboa, definindo as estradas, ruas e caminhos por onde os atores pedestres, carros ou trams se poderão movimentar.

Para representar um mapa no simulador, foi criada uma classe *Global_Map* que guarda a informação de um ou vários mapas, os necessários para correr a simulação. O simulador é assim capaz de fazer a leitura dos ficheiros mapa, e convertê-los para as suas estruturas de dados presentes na classe *Global_Map*. Este processo será melhor explicado no Capítulo 4, visto terem sido feitas modificações significativas nesta parte do simulador.

O simulador suporta dois tipos diferentes de mapas bem conhecidos, os Well-Known-Text e os Open Street Map.

4.1.5.1 *Well-Known-Text*

A primeira solução encontrada foi o uso de mapas Well-Known-Text, mapas estes já usados por outros simuladores de mobilidade, tal como o simulador *The One*. Este tipo de ficheiros respeitam regras de formatação

específicas, permitindo de forma simples definir distintas formas geométricas como pontos, linhas ou polígonos, bastando que para isso seja usada uma etiqueta e de seguida os respetivos dados. O Listagem 1 exemplifica a formatação de um ponto, e a de uma linha que passa por 3 pontos.

Listagem 1: Ficheiro Well-Known-Text

```
1 POINT (30 10)
2 LINESTRING (30 10, 10 30, 40 40)
```

Estes mapas são simples, gratuitos e multi-plataforma, enquadrando-se assim neste simulador.

4.1.5.2 *Open Street Map*

Outro formato de mapas suportado por este simulador, são os mapas Open Street Map (OSM), que são gratuitos, à escala mundial e elaborados pela comunidade OSM, permitindo assim que qualquer pessoa possa contribuir, mantendo os mapas atualizados. Estes mapas apresentam algumas diferenças face aos mapas WKT, trazendo algumas vantagens e melhorias, tornando-se assim mais completos. Para além da simples definição de pontos e linhas, este permite definir também semáforos, limites de velocidade, sinais de trânsito, ou até sentidos rodoviários, informação esta útil para uma simulação de mobilidade mais realista.

No site oficial[1] é possível visualizar e navegar nos mapas, e de forma simples exportar os mapas no formato OSM. Na Figura 28 pode-se ver um exemplo do excerto de um mapa, o qual podemos exportar.

A estrutura dos ficheiros OSM, respeitam regras de formatação para a definição de pontos, linhas ou outros elementos do mapa. Inicialmente são declarados todos os pontos, atribuindo-lhes um ID, e posteriormente estes são usados para definir linhas ou outros elementos do mapa, tais como semáforos ou paragens de autocarro. Uma linha trata-se então de um conjunto de pontos e é designada por *WAY*, à qual pode ser associada informação para definir se esta é uma estrada, passeio de peões ou até uma linha férrea. O Listagem 2 mostra a definição de um conjunto de pontos e de uma linha.

1 www.openstreetmap.org

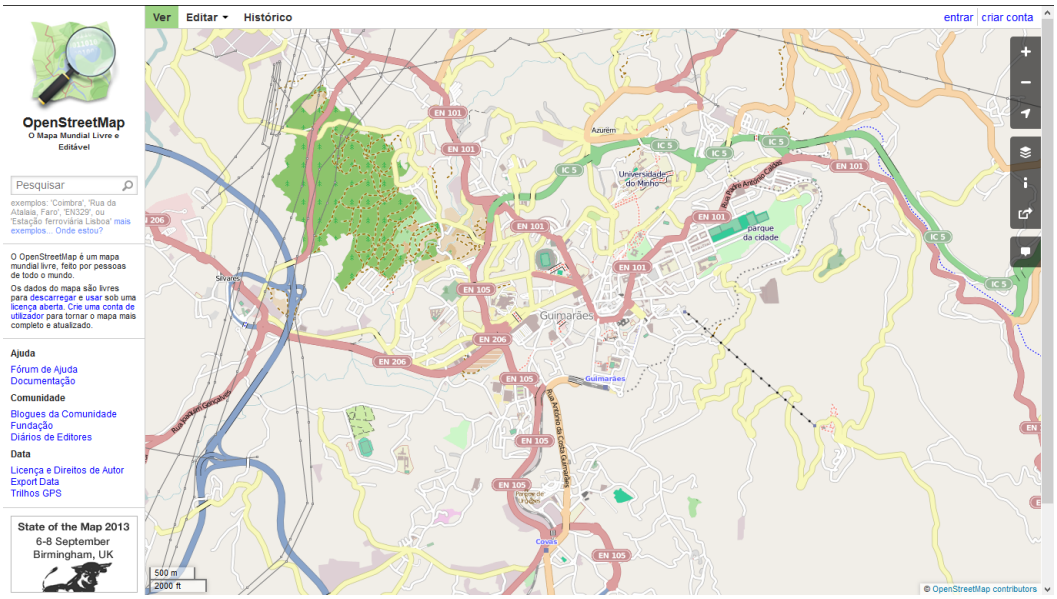


Figura 28: Mapas OSM

Listagem 2: Ficheiro OSM

```
<node id='-56573' visible='true' lat='41.5765925' lon='-8.363977' />
<node id='-56566' visible='true' lat='41.554385' lon='-8.354526' />
<node id='-56564' visible='true' lat='41.554635' lon='-8.354803' />
<node id='-56562' visible='true' lat='41.554968' lon='-8.355236' />
<node id='-56560' visible='true' lat='41.555382' lon='-8.355391' />
<way id='-56574' action='modify' visible='true'>
  <nd ref='-56573' />
  <nd ref='-56086' />
  <tag k='highway' v='trunk' />
</way>
```

4.1.6 *TCP Server*

O TCP Server é o elemento do simulador responsável por manter a comunicação entre *Global* e *Local Coordinators*, estando por isso a correr no *Global Coordinator*, esperando por novas conexões a pedido dos *Local Coordinators*.

É então usado para fazer o envio inicial dos mapas para os Local Coordinators, e posteriormente com o decorrer da simulação é usado para enviar ordens de criação de atores.

4.1.7 *TCP Client*

Este é o elemento presente nos *Local Coordinators* e *Vizualization*, que comunica com o *TCP Server* presente no *Global Coordinator*. É usado então pelos *Local Coordinators* para estabelecer uma conexão com o *Global Coordinator*, fazendo inicialmente o download dos mapas para a simulação e durante esta receber pedidos para geração de atores. Sendo que no Vizualization não existe qualquer geração de atores, esta ligação é usada unicamente para a recepção dos mapas da simulação.

4.1.8 *Multicast Sender*

Este elemento, tal como o *TCP Server* e *TCP Client*, é responsável por manter uma comunicação entre *Global Coordinator*, *Local Coordinator* e *Vizualization*, encontrando-se todos na mesma rede multicast. Sendo esta ligação usada para o envio das atualizações dos atores, apenas os *Local Coordinators* dispõem de um *Multicast Sender*. Foi usada uma ligação de multicast para fazer o envio das atualizações dos atores, visto esta facilitar o envio entre entidades no simulador, e visto que a perda de informação de uma atualização de um ator, não é critica.

4.1.9 *Multicast Receiver*

Este elemento é o que recebe a informação enviada pelo *Multicast Sender*, estando assim presente tanto nos *Local Coordinators* para que cada ator saiba a posição dos outros atores, como no *Global Coordinator* para que seja feito o reporting da movimentação e no *Vizualization* para que seja mostrada a movimentação dos atores no interface gráfico.

4.1.10 *Actors*

As entidades atores, são o ponto fulcral do simulador pois tudo está construído em redor deles, visto que será a partir das suas movimentações e comunicações, que serão feitos os estudos, investigações e testes resul-

tantes deste simulador.

Cada ator representa uma entidade da vida real, estando estes definidos em três tipos: pedestres, carros e trams. Cada tipo de ator, tem modelos de mobilidade distintos, procurando imitar da forma mais realista possível, o comportamento destes na vida real, tal como evitar choques entre eles, controlo da velocidade, etc...

4.1.11 *Generators*

Visto que o número de atores deve ser controlado, e estes criados também de forma controlada, os *Generators* são entidades responsáveis por criar atores. Existe um tipo diferente de *Generators* para cada tipo diferente de ator, os quais devem ser definidos nas configurações pré-simulação, consoante a necessidade. São ainda os *Generators*, os responsáveis por definir o ponto inicial de criação de atores no mapa, ou a velocidade inicial destes. A cadência de criação de atores, pode também ser configurada em cada *Generator*.

4.1.12 *Movement Reporting*

Este modulo é responsável por registar toda a movimentação dos atores. Durante a simulação, este módulo presente no *Global Coordinator* vai registando todos os movimentos efetuados pelos atores no mapa de simulação, permitindo assim que no fim da simulação, esta possa ser minuciosamente analisada.

O ficheiro de reporting gerado, poderá ser também usado pelo *Vizualization*, para mostrar de forma animada a simulação efetuada.

4.2 MELHORAMENTOS DE PERFORMANCE

4.2.1 *Descrição do Problema*

Com o começo da analise do simulador, foi depurado um ponto fraco no desempenho deste. Para a representação de um mapa, o simulador foi desenhado e construído de forma a usar uma classe *Global_Map*, onde era

criada uma lista de objetos que representam uma linha, para representar assim todas as linhas do mapa. Esta estrutura foi inspirada pelos mapas WKT (Well-known text), os quais são representados por várias listas de pontos, onde cada lista representa assim uma linha. A Figura 29 representa um mapa exemplo, com três diferentes linhas representadas por diferentes cores, e onde cada ponto é representado por $P_{(n)}$. A Figura 30 representa a lista de linhas relativamente ao mapa exemplo.

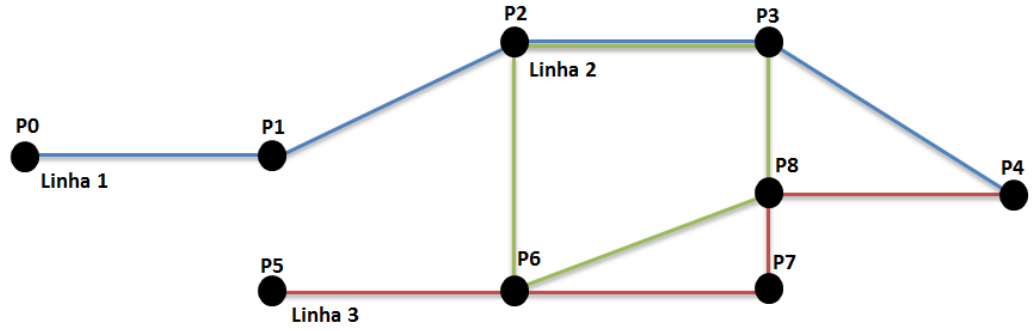


Figura 29: Mapa Exemplo

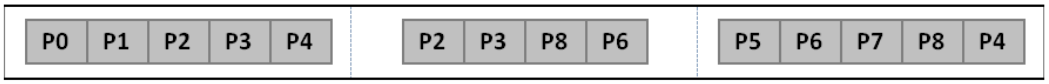


Figura 30: Estrutura do Mapa

Com o movimento dos atores pelo mapa, cada ator tem de usar esta estrutura para poder então movimentar-se entre pontos e escolher rotas. Assim a solução encontrada para o mecanismo de procura da próxima coordenada por parte do ator, passava por percorrer todas as linhas do mapa, e para cada linha percorrer todos os pontos, procurando assim por pontos anteriores ou seguintes ao ponto atual, para poder então criar uma lista de pontos possíveis para próximos destinos. Exemplificando, a Figura 31 mostra este mecanismo para o caso de estar a ser usado o mapa da Figura 29 e o ator se encontrar no ponto 2 (P2).

Esta técnica foi então tida em conta como muito penosa em termos de CPU para o simulador, pois cada vez que um ator pretende movimentar-se para outro ponto, teria então de pesquisar em todos os pontos do mapa por um próximo destino, quando em grande parte das vezes apenas exis-

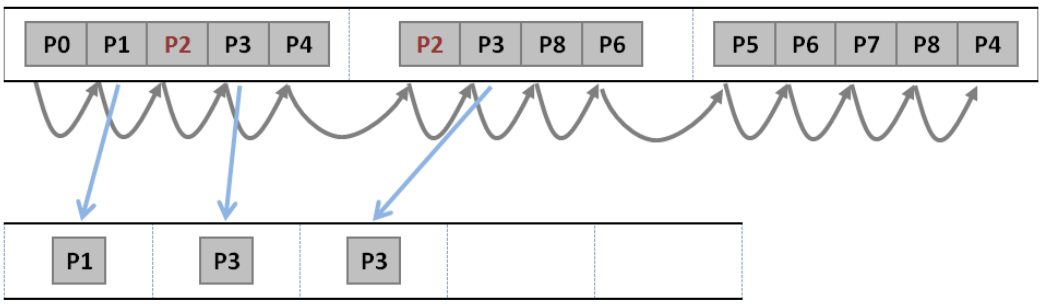


Figura 31: Procura de Destinos

tiria um ponto seguinte e um ponto anterior.

Foi então necessário pensar numa nova alternativa de representação do mapa, para solucionar este problema.

4.2.2 *Solução*

A solução encontrada para solucionar este problema passou pela substituição da estrutura que representa o mapa. Visto que grande parte do processamento passa pela necessidade de descobrir os possíveis pontos vizinhos de um ponto atual, surgiu naturalmente a ideia de armazenar para cada ponto os possíveis pontos vizinhos (destinos).

Foi então implementado um HashMap cuja chave é um valor identificativo do ponto e o valor uma lista de identificadores dos possíveis pontos destino. Desta forma quando um ator pretende mover-se para um novo destino, basta para isso que consulte a lista de pontos destino do ponto atual, e escolha um. A Figura 32 exemplifica a nova estrutura do mapa, para o mesmo exemplo do mapa da Figura 29.

Após a implementação desta nova estrutura de dados foi comparado o desempenho face à versão anterior. Para tal foi feito um teste em que um único ator executa a função de procura de novo destino um determinado número de vezes, e é contabilizado o tempo que este demora a executar as consecutivas chamadas à função.

Os testes foram efetuados com um elevado número de chamadas consecutivas à função, para que seja assim simples denotar as diferenças entre versões, neste caso 10.000 chamadas. Para cada versão foram ainda efetua-

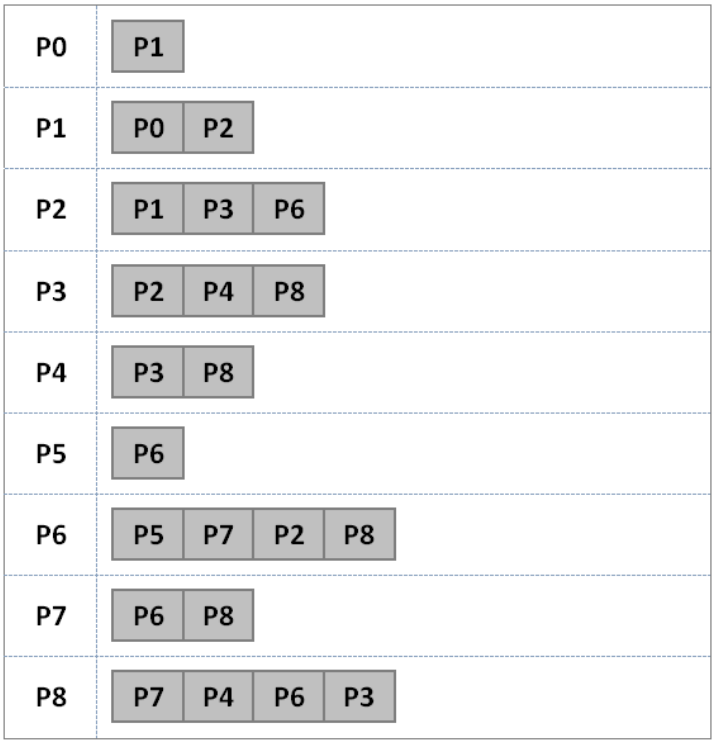


Figura 32: Estrutura Mapa: HashMap Pontos Destino

dos 10 testes, confirmando assim a coerência dos resultados, e que outros fatores não estariam a interferir nos tempos registados. Os resultados registados encontram-se então apresentados na Tabela 2 e na Tabela 3.

Como se pode concluir a partir dos resultados, a nova estrutura veio trazer uma melhoria muito significativa no tempo de processamento para a pesquisa de um próximo destino. Os testes realizados à função de pesquisa de próximo destino usando a estrutura de dados antiga, apresentam uma média de 10589 milissegundos, enquanto que com o uso da nova estrutura de dados HashMap apresentam apenas uma média de 69 milissegundos, resultando assim numa melhoria de aproximadamente -153 vezes menos tempo dispendido, no processamento do cálculo de um próximo destino.

Visto que para as chaves do HashMap são gerados valores únicos inteiros, identificativos dos pontos, foi pensado ainda substituir o HashMap por uma ArrayList, onde o índice do ArrayList corresponde ao identificador do ponto, e estaria nessa posição a lista de destinos tal como acontecia no HashMap. A Figura 33 mostra o ArrayList para o exemplo do mapa da Figura 29. Esta técnica traz como vantagem o acesso direto à lista de des-

Estrutura Antiga	
Teste	Tempo$_{(ms)}$
1	10.604
2	10.565
3	10.571
4	10.559
5	10.650
6	10.649
7	10.566
8	10.546
9	10.602
10	10.582
Média:	**10.589**

Tabela 2: Testes à Estrutura Antiga

Estrutura HashMap	
Teste	Tempo$_{(ms)}$
1	75
2	73
3	67
4	68
5	70
6	70
7	68
8	67
9	66
10	70
Média:	**69**

Tabela 3: Testes à Estrutura HashMap

tinos do ponto pretendido, visto que no HashMap teria sempre de existir uma procura pela chave.

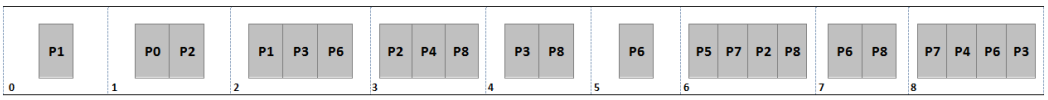


Figura 33: Estrutura Mapa: ArralyList Pontos Destino

Desta forma, foi então implementada a nova estrutura ArrayList, substituindo assim o HashMap com busca a melhorar ainda mais os tempos de processamento. Foram também feitos novos testes a esta estrutura, nas mesmas condições dos testes anteriores, e os resultados registados estão apresentados na Tabela 4.

Como se pode facilmente concluir pelos resultados registados ao uso desta nova estrutura, existiu ainda uma melhoria de 19% no tempo dispendido para o processamento de um próximo destino face à estrutura HashMap.

Estrutura Antiga	
Teste	Tempo$_{(ms)}$
1	10.604
2	10.565
3	10.571
4	10.559
5	10.650
6	10.649
7	10.566
8	10.546
9	10.602
10	10.582
Média:	**10.589**

Estrutura HashMap	
Teste	Tempo$_{(ms)}$
1	75
2	73
3	67
4	68
5	70
6	70
7	68
8	67
9	66
10	70
Média:	**69**

Estrutura ArrayList	
Teste	Tempo$_{(ms)}$
1	55
2	58
3	54
4	54
5	59
6	57
7	57
8	55
9	57
10	56
Média:	**56**

Tabela 4: Testes à Estrutura ArrayList

Assim foi definitivamente implementada esta solução, e foram feitos novos testes ao simulador, desta vez recorrendo a uma simulação normal, e não a testes consecutivos à função. Foi iniciada uma nova simulação com 2 geradores de atores com uma frequência de 2 atores por segundo durante 5 minutos, e registada a carga de CPU de 15 em 15 segundos. O gráfico da Figura 34 mostra a evolução da carga de CPU ao longo dos 5 minutos de simulação, permitindo assim comparar a diferença na carga de CPU com o uso da antiga e da nova estrutura de dados para representação do mapa.

Pode-se então concluir que foi alcançada com sucesso, uma grande melhoria no desempenho do simulador com o uso desta nova estrutura. Como se pode ver pelo gráfico da Figura 34, com a estrutura antiga o simulador no fim dos 5 minutos de simulação apresentava uma carga de CPU já muito elevada, próxima dos 100%. No entanto com o uso da nova estrutura ArrayList, o simulador ao fim dos 5 minutos apresenta apenas uma carga de CPU próxima dos 30%, respondendo assim de boa forma à simulação de aproximadamente 1200 atores.

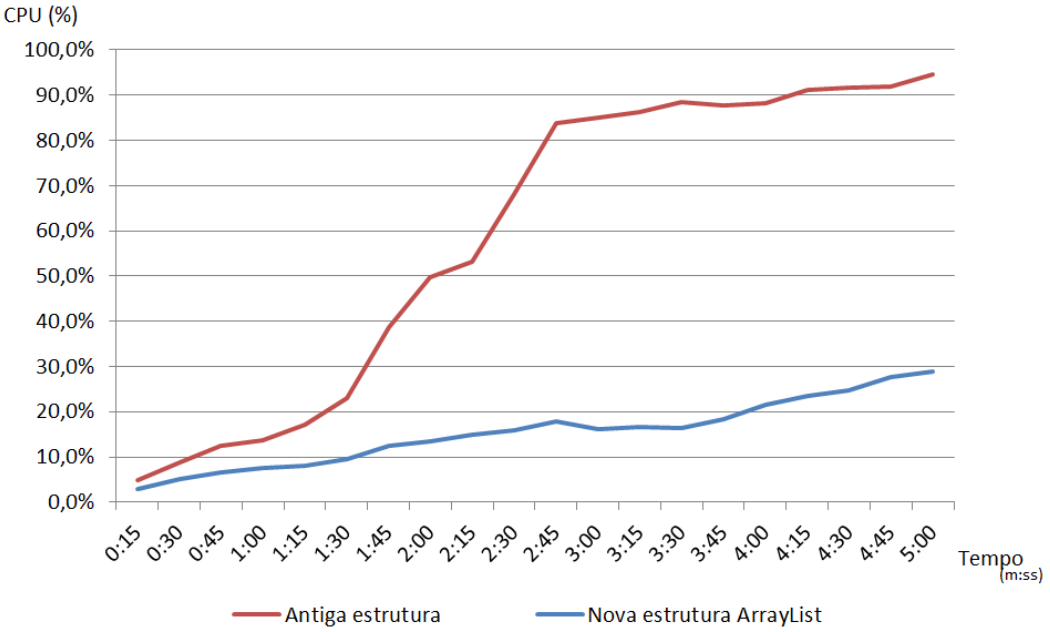


Figura 34: Gráfico Teste Desempenho

4.2.3 *Junção de Mapas*

O simulador foi pensado e desenhado para permitir o uso de diferentes mapas numa única simulação, tornando assim possível que diferentes tipos de atores, circulem num mapa específico ao seu tipo de mobilidade, tal como trams circularem num mapa de linhas de tram, carros num mapa de estradas e pedestres num mapa de ruas e passeios.

É ainda possível que um mesmo gerador de atores use em simultâneo mais de que um único mapa. Esta funcionalidade cria a hipótese de estender os mapas, isto é, imaginando que temos os mapas de todos os distritos de Portugal, podemos assim usa-los e correr a simulação num mapa de Portugal.

Enquanto que com a antiga estrutura de dados do mapa tal operação era simplificada, pois bastaria juntar todas as listas de linhas numa única lista, com a nova estrutura de dados baseada em destinos tal não é possível. Não poderão existir 2 pontos com as mesmas coordenadas, pois significaria destinos diferentes para cada um, e portanto os 2 mapas funcionariam em separado. Desta forma todos os pontos terão de ser únicos, e a eles associados os possíveis destinos provenientes de todos os possíveis diferentes mapas. Foi então elaborado um algoritmo para construir da forma

correta a junção dos mapas.

Para explicar melhor este procedimento, foi elaborado e desenhado um exemplo da junção de dois mapas. A Figura 35 mostra o mapa exemplo 1 e a respetiva estrutura de dados, a Figura 36 mostra o mapa exemplo 2 e a respetiva estrutura de dados. Estes mapas possuem 2 pontos com as mesmas coordenadas, o P6 do mapa 1 com o P1 do mapa 2, e o P7 do mapa 1 com o P3 do mapa 2. Por fim, a Figura 37 mostra a junção destes dois mapas e respetiva estrutura de dados resultante.

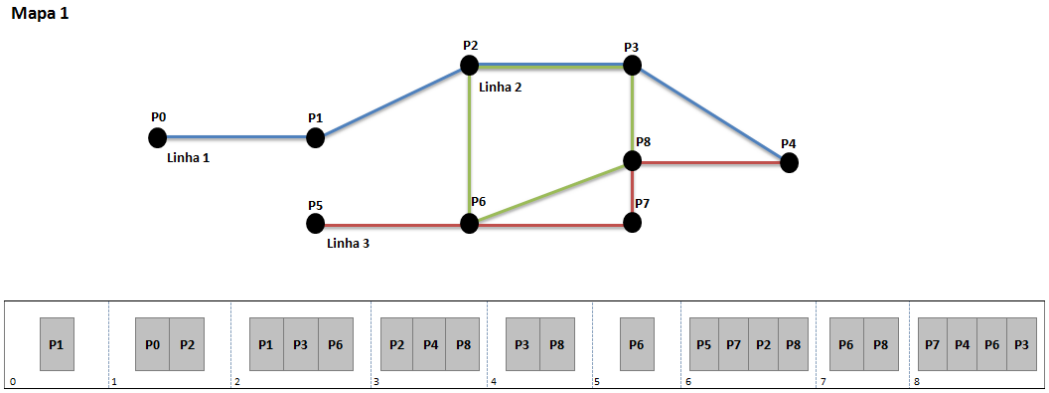


Figura 35: Mapa Exemplo 1

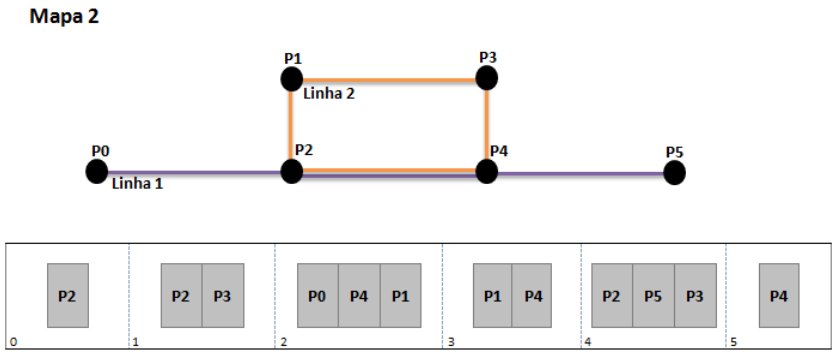


Figura 36: Mapa Exemplo 2

4.2.4 *Vizualization*

Com a implementação da nova estrutura de dados para a representação de um mapa, para que a antiga estrutura possa ser então descartada e excluída do simulador, foi necessário alterar o uso desta por parte do vizualization. Foi então construído um algoritmo para fazer o desenho

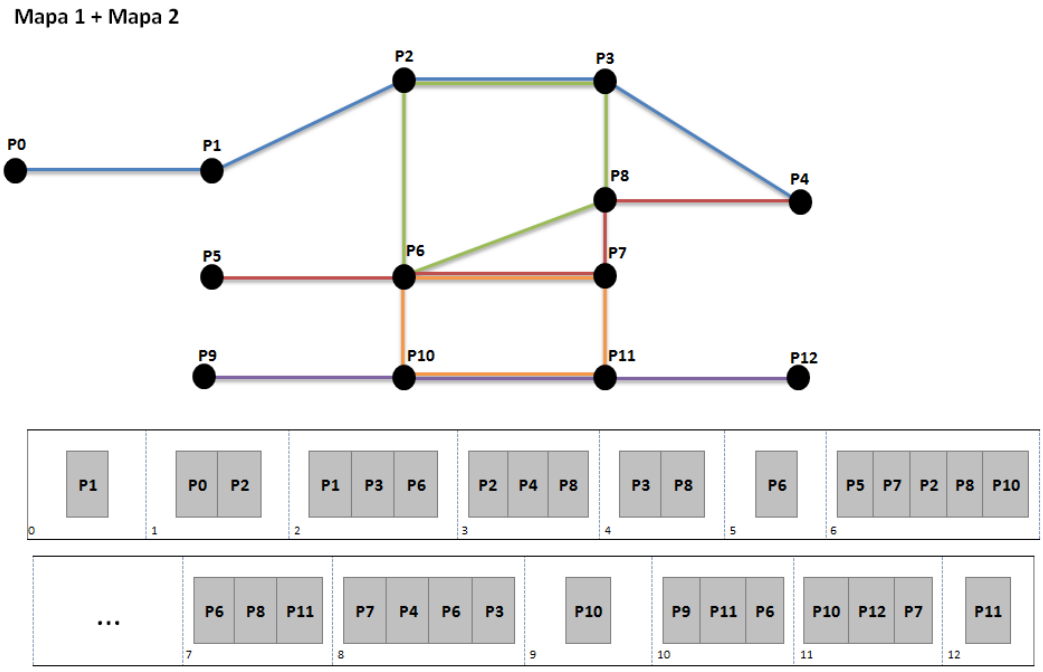


Figura 37: Mapa Exemplo 1 + Mapa Exemplo 2

do mapa baseado na nova estrutura de destinos e não na estrutura inicialmente usada baseada em linhas. O desenho do mapa passou então a residir na criação de linhas entre ponto e respetivos pontos destino.

4.2.5 *Mapas OSM*

Na alteração do código para a nova estrutura na parte de leitura de mapas OSM, foi detetado que o *parsing* dos mapas não esta a funcionar de forma genérica. Estes apenas eram carregados se os atributos das linhas estivessem todos presentes, e numa determinada ordem, desta forma era complicado usar os mapas descarregados diretamente do site oficial [2].

Foi então reprogramado o *parsing*, de forma a permitir ficheiros genéricos, com diferente número e ordem de atributos. Foram testados diferentes mapas, e todos foram carregados com sucesso.

2 http://www.openstreetmap.org/

4.3 CONCLUSÕES

Foi então contextualizada a arquitetura deste simulador, descrevendo e explicando os elementos que o compõe. Foram também explicados melhoramentos nele feitos, para combater alguns problemas de performance.

Com esta dissertação, tal como já foi dito anteriormente, pretende-se desenhar e implementar uma solução para a simulação de comunicações entre atores, simulando o protocolo *Bluetooth*. Pode-se ver assim na Figura 38 a arquitetura deste simulador com os elementos desenhados e implementados nesta dissertação, os quais serão explicados nos próximos capítulos.

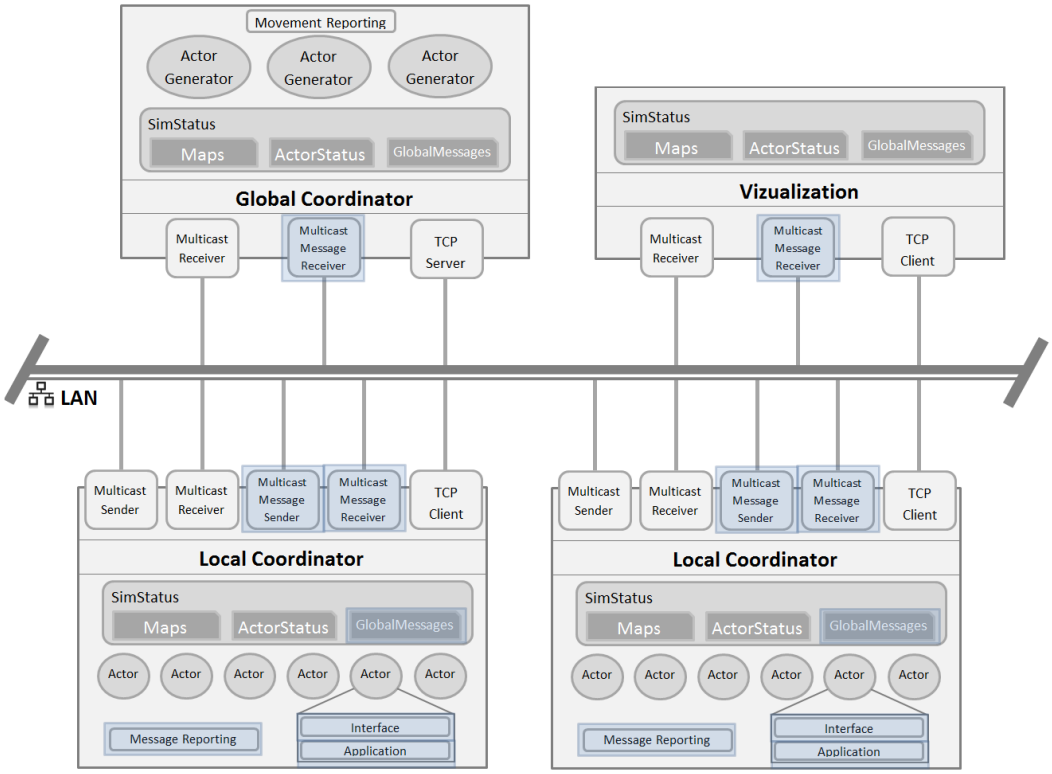


Figura 38: BartUM: Arquitetura

5

BARTUM - COMUNICAÇÕES

5.1 BARTUM - ARMAZENAMENTO DE MENSAGENS

Neste capítulo serão descritos os problemas de integração das comunicações no simulador, bem como as soluções encontradas e respetivas implementações efetuadas.

5.1.1 *Descrição do Problema e Abordagem*

Após o simulador ser então capaz de simular movimento de diversos tipos de atores, funcionando de forma distribuída e disponibilizando uma interface gráfica para a visualização de resultados, é agora necessário desenhar e implementar mecanismos para a simulação das comunicações entre atores, e simular um interface de comunicação específico, neste caso o Bluetooth.

Estando este simulador num âmbito de simulação da vida-real, a comunicação entre atores segue o mesmo caminho devendo para tal procurar imitar de forma realista os modelos de comunicações dos dispositivos móveis, procurando oferecer tanto as mesmas funcionalidades como reproduzir o mesmo funcionamento.

Há diversos fatores que deverão ter sidos em conta para fazer tal simulação, a movimentação dos atores no mapa é um deles pois influência o sucesso de uma conexão, sendo que num determinado instante de tempo estes podem conseguir comunicar mas no instante de tempo seguinte já não. Tal acontece devido a outro fator que é o alcance da tecnologia em questão. O simulador poderá dispor de várias tecnologias de comunicação tal como Wi-Fi, ZigBee, WAVE ou Bluetooth, e dependendo da tecnologia poderão assim existir diferentes raios de cobertura rádio.

Para implementação destas tecnologias de comunicação, terá de existir no simulador um sistema de armazenamento e distribuição de mensagens entre atores, que terá de ser independente da tecnologia servindo assim

para qualquer uma, permitindo que estas possam ser implementadas por cima deste sistema de armazenamento e distribuição de mensagens, funcionando como módulos inseridos no simulador.

Para tal foi necessário pensar em mecanismos e métodos para o armazenamento e distribuição das mensagens, tendo em atenção diversos fatores tais como o desempenho, recursos necessários e o estado atual do simulador, para que estes métodos possam ser integrados no sistema já existente. Foram pensadas e encontradas algumas possíveis soluções para o armazenamento e distribuição das mensagens, para as quais foram feitos protótipos com o fim de avaliar o seu desempenho. Cada protótipo foi construído a partir da mesma base de funcionamento, para que pudessem ser comparados única e exclusivamente tendo em conta as diferenças no método de armazenamento e distribuição de mensagens.

Foi para tal, criada uma classe Node que representa um ator, que corresponde a uma thread iniciada quando o utilizador começa a simulação. Cada ator começa por definir a sua posição inicial de forma aleatória, respeitado os limites do espaço de simulação. De seguida entra num ciclo infinito, onde faz repetidamente os seguintes passos:

1. Move-se num raio definido, respeitado os limites de espaço.
2. Gera uma mensagem para um outro ator escolhido aleatoriamente.
3. Trata das mensagens geradas e das mensagens recebidas, consoante a solução escolhida.
4. Adormece por um intervalo de tempo definido pelo utilizador.

Para facilitar a realização dos testes, foi também criado um interface gráfico (Figura 39) onde o utilizador poderá definir o número de atores na simulação, intervalo de tempo entre ciclos de cada ator, escolher se o ator guarda as mensagens recebidas numa lista pessoal ou não e visualizar também alguns dados como o número de mensagens geradas, número de mensagens recebidas por um determinado ator, etc.

Para avaliar o desempenho e viabilidade das soluções encontradas, foram feitos e registados testes ao protótipo. Os testes consistem em correr o protótipo com um determinado número de atores, e avaliar a carga de CPU, memória usada e número de mensagens geradas, em determinados

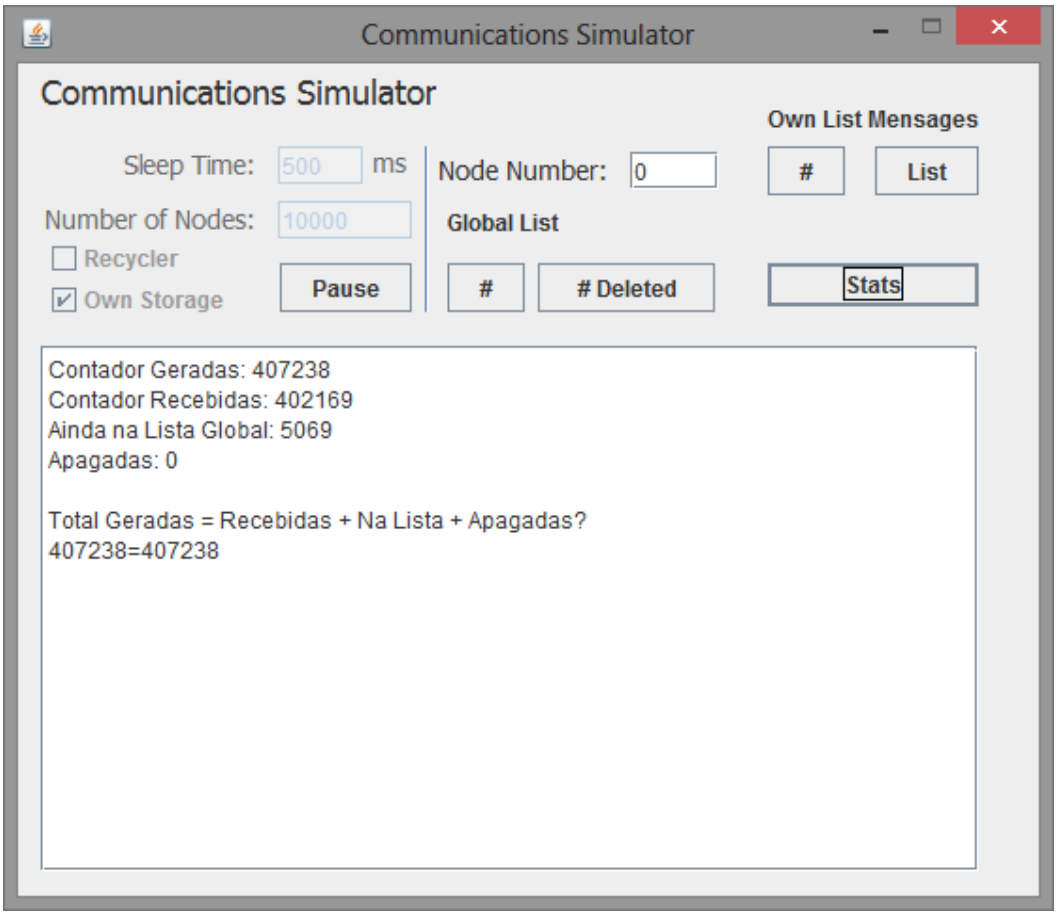


Figura 39: Envio de Mensagens: Protótipo

intervalos de tempo. Com os resultados dos testes será possível avaliar e comparar o desempenho entre as várias soluções.

A carga de CPU é um fator fulcral para a análise, pois como o simulador é um projeto em grande escala, é importante garantir que este seja o mais eficiente e estável possível para um bom funcionamento global. O uso de um método mais gastador de CPU nesta parte de envio de mensagens, seria então penoso para todo o simulador.

Pelas mesmas razões de fiabilidade de todo o simulador, é necessário controlar e gerir da forma mais eficiente o consumo de memória. Esta parte de envio de mensagens terá de ter um cuidado especial com o consumo de memória, pois o número de mensagens geradas e consequente método de armazenamento poderá ser muito elevado, contribuindo desta forma para um grande aumento da memória usada pela aplicação.

É também importante controlar o número de mensagens geradas num determinado intervalo de tempo, pois como foi verificado há soluções com uma baixa eficiência, isto é, o número de mensagens geradas fica muito aquém do idealmente esperado.

5.1.2 *Solução 1 - Lista Global de Mensagens*

5.1.2.1 *Descrição*

Esta solução, talvez por ser a mais óbvia, foi a primeira solução encontrada. Consiste numa lista global de mensagens partilhada e acessível a todos os atores, que dessa forma poderão adicionar à lista tanto as mensagens geradas por si para outros atores, tal como retirar da lista as mensagens para si vindas de outros atores. A Figura 40 ilustra o funcionamento desta solução.

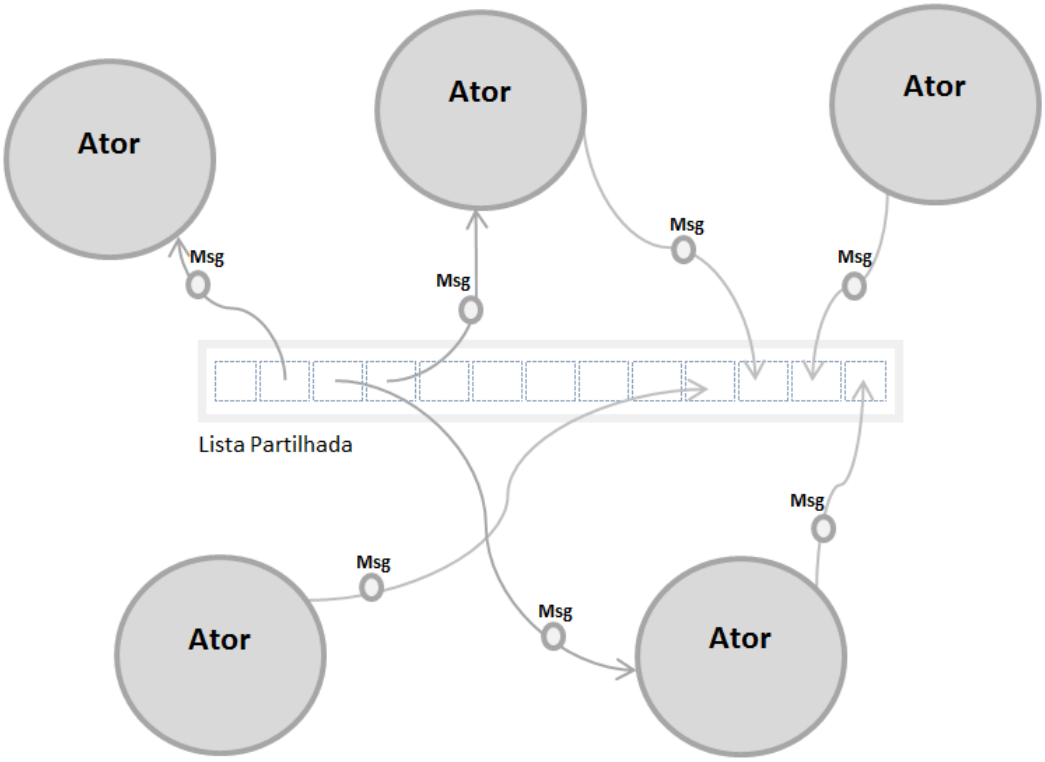


Figura 40: Envio de Mensagens: Solução 1

5.1.2.2 *Implementação*

A implementação deste método é bastante simples, com uma classe pública "Global", é declarada uma lista de mensagens como se pode ver no

Listagem 3. Desta forma cada thread ator terá de colocar na lista as mensagens geradas por si para outros atores, tal como retirar da lista as mensagens que são para si.

Listagem 3: Declaração: Lista Global de Mensagens

```
1 public static ArrayList<Msg> lmsg = new ArrayList<Msg>();
```

Para que este processo funcione da forma esperada, é necessário controlar a concorrência no acesso à lista global de mensagens para que, por exemplo, dois ou mais atores não insiram mensagens diferentes na mesma posição da lista devido ao acesso simultâneo desta, o que resultaria em mensagens perdidas.

Para colmatar este problema foi então usado o método *synchronized*, que garante a exclusividade no acesso à lista global de mensagens. Uma vez bloqueada a lista global de mensagens para um determinado ator, este insere-as na lista, e já que a tem bloqueada verifica também se nela existem mensagens destinadas a si, para assim as retirar da lista. O Listagem 4 mostra esta operação.

Listagem 4: Acesso à Lista Global de Mensagens

```
1 synchronized(Global.lmsg){
2         Global.lmsg.add(msg);
3         for(int i = 0; i<Global.lmsg.size();i++){
4                 if(Global.lmsg.get(i).getNodeidD()==this.nodeid){
5                         this.lmsg.add(Global.lmsg.get(i));
6                         Global.lmsg.remove(i);
7                 }
8         }
9 }
```

5.1.2.3 *Integração no Simulador*

Para integrar esta solução no simulador, seria necessário um espaço de memória partilhado por todos os atores em cada *Local Coordinator*, onde se poderia tirar proveito da já existente classe *SimStatus*. Seria também necessário criar um sistema de sincronização das listas para que as mensagens fossem dispersadas por todos os atores de uma simulação que se encontram dispersos pelos vários *Local Coordinators*.

Para a implementação de mensagens *Broadcast* bastaria criar uma entrada na lista global de mensagens, dirigida às mensagens Broadcast, onde todos os atores a poderiam lá colocar e ler as mensagens Broadcast.

5.1.2.4 *Testes*

Para avaliar a performance desta solução, foi então realizada uma monitorização ao comportamento do seu protótipo durante a execução. Foi efetuada uma simulação com uma duração de 30 minutos e 20.000 atores (população da cidade de Barcelos) a enviarem uma mensagem e a receberem as mensagens destinadas a si, a cada intervalo de 500 milissegundos. Nestas condições a situação ideal seria serem geradas 2.400.000 mensagens após 1 minuto, 24.000.000 mensagens após 10 minutos e 72.000.000 mensagens após 30 minutos. Foi usado um computador com um processador i7 3610QM, 6Gb de memória RAM e com o sistema operativo Windows 8 64 bits. Os resultados desta simulação foram os seguintes:

	CPU	Memória	Mensagens Geradas
1 minuto	19,6%	1539,5 Mb	36.247
10 minutos	19,4%	1266,7 Mb	275.440
30 minutos	19,5%	1262,8 Mb	812.790

Tabela 5: Envio de Mensagens: Solução 1 - Testes

Os resultados destacam o facto de esta solução não ser eficiente, gerando um número de mensagens muito baixo, cerca de 1,5% do valor ideal esperado. Tal acontece devido ao uso de um único ArrayList global para todos atores, e à sua alta concorrência de acesso, fazendo com que num determinado instante de tempo apenas possa estar um único ator a escrever ou a retirar as suas mensagens da lista, bloqueado assim todos os outros e provocando uma enorme fila de espera.

Outro aspeto a denotar é a carga de CPU que se mantém estável ao longo do tempo, contudo bastante elevada quando comparada com as outras soluções que iremos ver a seguir.

5.1.3 *Solução 2 - Lista Global de Mensagens Distribuída por Atores*

5.1.3.1 *Descrição*

Esta solução é idêntica à solução anterior, consistindo de igual forma numa lista global de mensagens onde cada ator insere as mensagens geradas por si para outros atores, e retira as mensagens destinadas a si geradas por outros atores.

A grande diferença perante a solução anterior, é que a lista global de mensagens estar distribuída por ator, passando cada ator a ter uma lista de mensagens correspondente, identificada pelo seu ID. Esta solução permite que o sincronismo seja distribuído, não sendo necessário bloquear toda a lista de mensagens, mas apenas a "sub-lista"de cada ator. A Figura 41 ilustra esta solução.

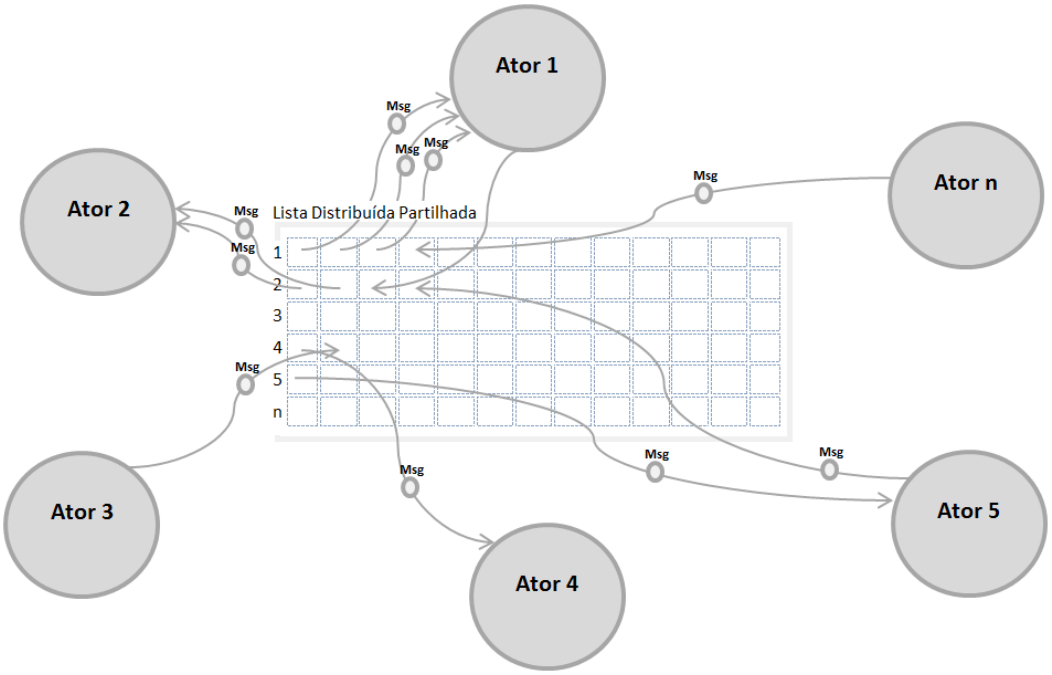


Figura 41: Envio de Mensagens: Solução 2

5.1.3.2 *Implementação*

Para implementar esta solução foi declarada na classe pública Global um HashMap onde a chave é o ID do ator e o conteúdo a lista de mensagens desse ator, tal como mostra o Listagem 5. Assim a lista global fica

distribuída por ator, onde em vez de um ArrayList de mensagens para todos os atores, existe um ArrayList de mensagens para cada ator, tal como mostra a Figura 41.

Listagem 5: Declaração: Lista Global de Mensagens Distribuída - HashMap

```
1 public static HashMap lm=new HashMap<Integer,ArrayList<Msg>>();
```

Foi ainda pensado o uso de outros métodos de armazenamento tal como o ConcurrentHashMap ou o HashTable em vez do tradicional HashMap, pois estes métodos já garantem sincronismo nos processos de escrita e remoção de elementos. Contudo, como o processo de procura de mensagens engloba também a remoção destas, estas duas operações têm de ser executadas em bloco, não bastando portanto um mecanismo de armazenamento que garanta o sincronismo, pois continua a existir a necessidade de usar o método synchronized para a execução destes dois passos, tal como mostra o Listagem 6.

Listagem 6: Solução 2: Procura de Mensagens

```
1 synchronized(Global.lm.get(this.nodeid)){
2         if(!((ArrayList<Msg>) Global.lm.get(this.nodeid)).isEmpty()){
3                 this.lmsg.addAll(((ArrayList<Msg>) Global.lm.get(this.
                      nodeid)));
4                 ((ArrayList<Msg>) Global.lm.get(this.nodeid)).clear();
5         }
6 }
```

Este método tem a vantagem perante o método anterior, de que quando o ator com o ID x pretende enviar uma mensagem para o ator com o ID y, apenas necessita de bloquear o ArrayList do ator y, e para retirar as suas mensagens da lista apenas necessita de bloquear o ArrayList do seu próprio ID, evitando desta forma ter de bloquear uma lista global inteira, onde existe uma alta concorrência de acesso.

5.1.3.3 *Integração no Simulador*

Para integrar esta solução no simulador, da mesma forma que na solução anterior seria necessário um espaço de memória partilhado por todos

os atores em cada *Local Coordinator*, onde se poderia tirar proveito da já existência da classe *SimStatus*. Seria também necessário criar um sistema de sincronização das listas para que as mensagens fossem dispersadas por todos os atores de uma simulação que se encontram dispersos pelos vários *Local Coordinators*.

Para a implementação de mensagens *Broadcast* bastaria criar uma entrada na lista global de mensagens, dirigida às mensagens Broadcast, onde todos os atores a poderiam lá colocar e ler as mensagens Broadcast.

5.1.3.4 *Testes*

Tal como na solução anterior foi então realizada uma monitorização ao comportamento do seu protótipo durante a execução. Foi feita uma simulação com uma duração de 30 minutos e 20.000 atores, a enviarem uma mensagem e a receberem as mensagens destinadas a si a cada intervalo de 500 milissegundos. Nestas condições a situação ideal seria serem geradas 2.400.000 mensagens após 1 minuto, 24.000.000 mensagens após 10 minutos e 72.000.000 mensagens após 30 minutos. Foi usado um computador com um processador i7 3610QM, 6Gb de memória ram e com o sistema operativo Windows 8 64 bits. Os resultados desta simulação foram os seguintes:

	CPU	Memória	Mensagens Geradas
1 minuto	5,4%	1506,3 Mb	2.372.340
10 minutos	5,8%	1020,2 Mb	23.959.276
30 minutos	5,3%	882,8 Mb	71.875.608

Tabela 6: Envio de Mensagens: Solução 2 - Testes

Os resultados demostram que o número de mensagens geradas é muito próximo do número ideal, e com tendencia a aproximar-se. Isto possivelmente acontece, devido ao facto de haver atraso no inicio da simulação no processo de criação das threads, e atrasar assim a geração de mensagens no arranque da simulação. Contudo o valor aproxima-se do ideal ao longo do tempo, o que demonstra que possivelmente não existirá qualquer atraso durante o processamento normal da simulação.

Pode-se também concluir que a carga de CPU é baixa e estável ao longo do tempo.

5.1.4 *Solução 3 - Lista Global de Mensagens Distribuída por Espaço*

5.1.4.1 *Descrição*

Esta solução é baseada na ideia de uma lista global de mensagens distribuída, tal como na solução anterior, porém com a diferença que a lista não está distribuída por atores, mas sim por espaço do mapa.

Como a simulação decorre num determinado espaço, e os atores têm um comportamento de movimento inspirado na vida real, não existe portanto a possibilidade de teletransporte dos atores, isto é, estes circulam pelo mapa sempre seguindo um determinado caminho a uma determinada velocidade, sem possibilidade de alterarem a sua posição instantaneamente. Assim foi possível imaginar uma solução com uma lista de mensagens partilhada por espaço, isto é, quando um ator pretende enviar uma mensagem para outro ator este coloca a mensagem na lista de mensagens correspondente à área de espaço em que se encontra. Por outro lado quando um ator pretende receber as mensagens destinadas a si terá de as procurar apenas na lista de mensagens da área em que se encontra e nas vizinhas, devido à possibilidade de movimento por parte dos atores. Como exemplifica a Figura 42, um ator que se encontre na área 46 do mapa, guarda as suas mensagens na lista da área 46. E quando pretende ver se tem mensagens para si, consulta para além da lista da área da posição onde se encontra, as listas de mensagens das áreas vizinhas.

Esta solução imita de melhor forma o funcionamento tal como na vida real, pois como qualquer comunicação por rádio tem a si associado um alcance espacial, cada ator só receberá mensagens da própria área ou de áreas vizinhas.

5.1.4.2 *Implementação*

Esta é a solução com maior complexidade de implementação. Foi primeiro necessário criar um método para dividir o mapa por áreas, e associar uma lista de mensagens a cada área. Para tal foi definida uma cons-

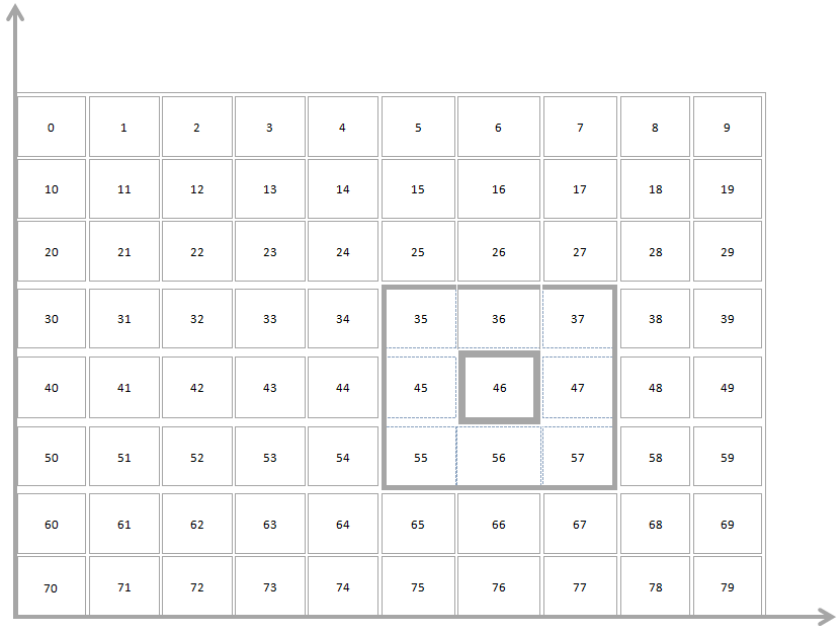


Figura 42: Envio de Mensagens: Solução 3 (Mapa)

tante com o valor de divisão, criando subdivisões quadradas no mapa, traduzindo-se assim o número total de áreas por:

$$\text{NúmerodeÁreas} = \frac{\text{LarguradoMapa}}{\text{ConstantedeDivisão}} * \frac{\text{AlturadoMapa}}{\text{ConstantedeDivisão}}$$

É necessário ter em atenção a possibilidade do valor não ser inteiro, e ser então necessário fazer um arredondamento para cima, e com este valor são então criadas as listas para cada sub-área do mapa.

Com o mapa dividido, é necessário que cada ator consiga então identificar a área em que se encontra no mapa, para que possa escrever as suas mensagens, e procurar mensagens para si. Para isto terá de fazer o seguinte cálculo:

$$\text{ÁreaActual} = \left(\text{trunc}\left(\frac{x}{\text{ConstantedeDivisão}}\right) * \frac{\text{LarguradoMapa}}{\text{ConstantedeDivisão}}\right) + \text{trunc}\left(\frac{y}{\text{ConstantedeDivisão}}\right),$$

onde x representa a posição atual do ator no eixo das abscissas, e y a posição atual do ator no eixo das ordenadas. A função trunc, representa a função de arredondamento para baixo. O Listagem 7 mostra a implementação deste cálculo.

Listagem 7: Cálculo da área atual

```
1 numlista = ((((int) Math.floor(this.x/GRAPHDIV)) * (POSMAX/GRAPHDIV)))
      + (int) Math.floor(this.y/GRAPHDIV);
```

Após o ator descobrir em que área se encontra, poderá então guardar as mensagens geradas por si na lista de mensagens correspondente a essa área. Para isso, deverá bloquear a lista de mensagens dessa área antes de escrever a mensagem, devido à possível concorrência ao acesso desta lista, evitando assim uma possível perda de mensagens.

Para retirar as mensagens de outros atores destinadas a si, tal como já foi dito o ator apenas necessita de procurar na lista de mensagens da área em que se encontra, e nas listas de mensagens das áreas vizinhas. É então necessário em primeiro lugar analisar quais as áreas vizinhas da área em que se encontra.

Um ator poderá ter um número variável de vizinhos, este número poderá ser 0 caso apenas haja uma área, poderá ser 1 caso existam apenas 2 áreas, poderá ser 3 caso se encontre num canto do mapa, poderá ser 5 caso se encontre numa "parede"do mapa ou poderá ser 8 caso se encontre numa área no interior do mapa.

Para fazer esta verificação foram usados dois métodos. O primeiro verifica se a área atual do ator é múltipla da $\frac{\mathrm{LarguradoMapa}}{\mathrm{ConstantedeDivisão}}$, se for significa que o ator se encontra numa área da "parede"da esquerda do mapa, ou então se a área atual do ator + 1, é também múltipla da $\frac{\mathrm{LarguradoMapa}}{\mathrm{ConstantedeDivisão}}$, neste caso significando que se encontra numa área pertencente à parede direita do mapa. Para cada um destes casos, ou para o caso de nenhuma destas situações se verificar, existem posições definidas das áreas vizinhas que serão então tidas em conta para a procura de mensagens. O outro método para a escolha dos vizinhos é usado após as verificações anteriores, desta vez verificando se as listas dadas como vizinhas existem. Caso alguma não exista significa que a área atual está num canto do mapa.

Após saber então quais as áreas vizinhas da área da posição atual, é feita a procura de mensagens nas respetivas listas. É também necessário bloquear a lista de mensagens devido à possível concorrência ao acesso desta lista, evitando assim uma possível perda de mensagens.

5.1.4.3 *Integração no Simulador*

Para integrar esta solução no simulador, visto que esta também se baseia numa lista de mensagens partilhada, seria necessário existir um espaço de memória partilhado por todos os atores em cada *Local Coordinator*, onde se poderia tirar proveito da já existência da classe *SimStatus*. Seria também necessário criar um sistema de sincronização das listas para que as mensagens fossem dispersadas por todos os atores de uma simulação que se encontram dispersos pelos vários *Local Coordinators*.

Para a implementação de mensagens *Broadcast* bastaria criar uma entrada na lista global de mensagens, dirigida às mensagens Broadcast, onde todos os atores a poderiam lá colocar e ler as mensagens Broadcast.

5.1.4.4 *Testes*

Tal como nas soluções anteriores foi então realizada uma monitorização ao comportamento do seu protótipo durante a execução. Foi feita uma simulação durante 30 minutos com 20.000 atores, a enviarem uma mensagem e a receberem as mensagens destinadas a si a cada intervalo de 500 milissegundos. Nestas condições a situação ideal seria serem geradas 2.400.000 mensagens após 1 minuto, 24.000.000 mensagens após 10 minutos e 72.000.000 mensagens após 30 minutos. Foi usado um computador com um processador i7 3610QM, 6Gb de memória RAM e com o sistema operativo Windows 8 64 bits. Os resultados desta simulação foram os seguintes:

	CPU	Memória	Mensagens Geradas
1 minuto	99,8%	1473,2 Mb	82.820
10 minutos	99,3%	1488,8 Mb	193.899
30 minutos	99,7%	1598,5 Mb	278.361

Tabela 7: Envio de Mensagens: Solução 3 - Testes

O número de mensagens geradas ficou muito aquém das expetativas, sendo apenas de 3,45% do número ideal ao fim do primeiro minuto, e diminuindo sempre até 0,39% ao fim de 30 minutos de execução.

A carga de CPU manteve-se sempre na volta dos 100%, significando que esta solução é inviável devido ao excesso de processamento provocado pelos cálculos de coordenação no mapa.

5.1.5 *Solução 4 - Escrita Direta das Mensagens*

5.1.5.1 *Descrição*

Durante a construção dos protótipos e respetiva preocupação com os recursos usados, foi também pensada uma solução onde não existe um espaço partilhado para a distribuição das mensagens, mas sim a escrita direta destas no ator destinatário.

Desta forma, quando um ator pretende enviar uma mensagem para outro ator, basta que a escreva diretamente na lista de mensagens recebidas de cada ator, portanto na própria classe deste, tal como demonstra a Figura 43.

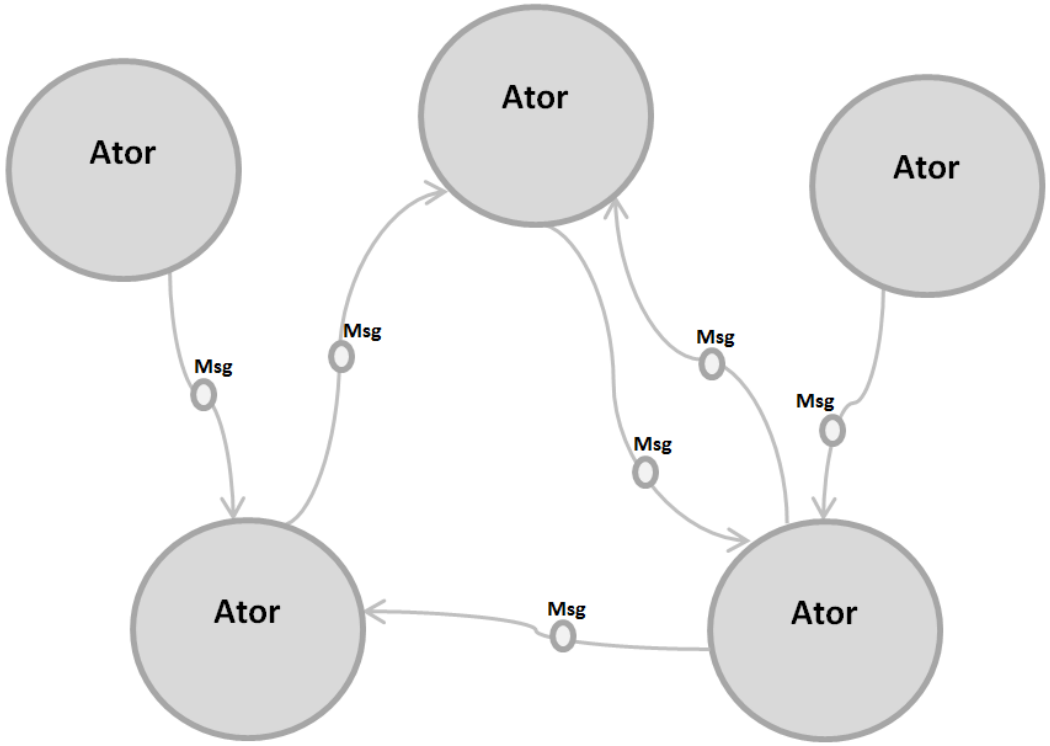


Figura 43: Envio de Mensagens: Solução 4

5.1.5.2 *Implementação*

A implementação deste método é a mais simples de todas, pois não necessita de qualquer esquema de armazenamento partilhado, poupando recursos de memória e processamento. Contudo, esta solução tem a necessidade de que haja acesso direto aos atores, portanto que o apontador para estes seja público, para que seja então possível que cada ator tenha acesso direto a qualquer outro ator.

Com este método continua a ser necessário bloquear a lista de mensagens recebidas de cada ator quando se pretende lá inserir uma mensagem pois poderá existir concorrência entre atores que gerem mensagens para um mesmo ator no mesmo espaço de tempo. O Listagem 8 mostra a simples implementação deste método.

Listagem 8: Escrita das Mensagens diretamente no Ator

```
1 synchronized(Global.lnode.get(nodeidd).lmsg){
2         Global.lnode.get(nodeidd).lmsg.add(msg);
3 }
```

5.1.5.3 *Integração no Simulador*

Para integrar esta solução no simulador, não seria necessário existir um espaço de memória partilhado por todos os atores visto que estes escrevem de forma direta no ator destino. Contudo esta técnica levanta o problema do envio de mensagens para atores que se encontram em *Local Coordinators* distintos. Para que fosse possível tal técnica, seria necessário recorrer ao uso de um *Remote Direct Memory Access* (RDMA), sendo este um mecanismo que permite que via interface de rede, um computador consiga escrever informação diretamente na memória de outro computador.

Seria também complexa a implementação de mensagens *Broadcast*, visto não existir um espaço de memória partilhada entre os diversos atores. A solução passaria pela conversão de uma mensagem *Broadcast* em diversas mensagens *Unicast*, dependendo do número de atores na simulação.

5.1.5.4 *Testes*

Foi então realizada uma monitorização ao comportamento do seu protótipo durante a execução. Foi feita uma simulação com uma duração de 30 minutos e 20.000 atores, a enviarem uma mensagem e a receberem as mensagens destinadas a si a cada intervalo de 500 milissegundos. Nestas condições a situação ideal seria serem geradas 2.400.000 mensagens após 1 minuto, 24.000.000 mensagens após 10 minutos e 72.000.000 mensagens após 30 minutos. Foi usado um computador com um processador i7 3610QM, 6Gb de memória ram e com o sistema operativo Windows 8 64 bits. Os resultados desta simulação foram os seguintes:

	CPU	Memória	Mensagens Geradas
1 minuto	1,7%	1440,8 Mb	2.346.692
10 minutos	2,0%	1011,8 Mb	23.883.413
30 minutos	2,7%	872,7 Mb	71.807.074

Tabela 8: Envio de Mensagens: Solução 4 - Testes

Tal como na segunda solução o número de mensagens geradas é muito próximo do ideal e com tendencia a aproximar-se. É também bastante evidente a baixa carga de CPU durante a simulação tal como previsto, pois como a escrita das mensagens é direta no destinatário não existe a necessidade de qualquer procura destas.

5.1.6 *Solução 5 - Classe Distribuidora de Mensagens*

5.1.6.1 *Descrição*

Outra solução encontrada foi a possibilidade de atribuir as tarefas de inserir e retirar as mensagens da lista de mensagens, a uma classe gestora de mensagens, em vez de serem os atores a ter essa responsabilidade.

Desta forma quando um ator pretende enviar uma mensagem para um outro ator, este não tem de se preocupar em como o fazer, bastando para isso usar os métodos desta classe gestora que se encarregará de fazer a gestão das mensagens, tal como ilustra a Figura 44.

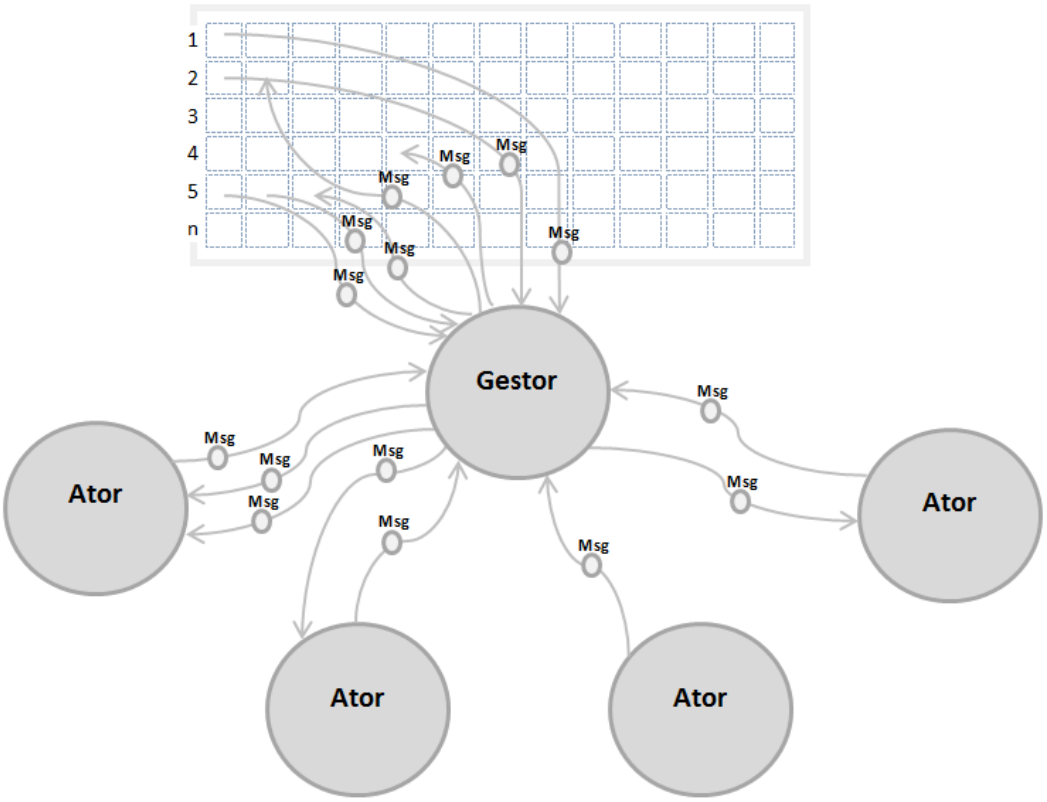


Figura 44: Envio de Mensagens: Solução 5

5.1.6.2 *Implementação*

Para a implementação desta solução, foi então necessário criar uma nova classe para ser a gestora das mensagens. Esta nova classe teria de fornecer aos atores dois serviços, guarda as mensagens por estes geradas e entregar-lhes as mensagens geradas por outros atores. Foram então implementados dois métodos, o método addMsg e o método getMsgs, e foi mantida a lista global de mensagens distribuída por ator, sendo nela que esta classe armazena as mensagens.

O primeiro método addMsg recebe por parâmetro uma mensagem e tem a função de a guardar na lista. Para tal analisa a mensagem para descobrir quem é o destinatário, e coloca-a na lista global distribuída.

O outro método getMsgs recebe por parâmetro o ID do ator e devolve uma lista com todas as mensagens que estão armazenadas cujo destinatário é esse ator.

Assim, não existe para cada ator a necessidade de se preocupar com o armazenamento das mensagens, bastando recorrer a estes dois métodos da classe gestora de mensagens, para enviar e receber as respetivas mensagens, tal como mostra o Listagem 9.

Listagem 9: Uso da classe gestora (air) pelos atores

```
1 //Enviar Mensagem
2 Global.air.addMsg(msg);
3 //Receber as minhas Mensagens
4 this.lmsg.addAll(Global.air.getMsgs(this.nodeid));
```

5.1.6.3 *Integração no Simulador*

Para integrar esta solução no simulador, visto que esta também se baseia numa lista de mensagens partilhada, seria necessário existir um espaço de memória partilhado por todos os atores em cada *Local Coordinator*, onde se poderia tirar proveito da já existência da classe *SimStatus*. Seria também necessário criar um sistema de sincronização das listas para que as mensagens fossem dispersadas por todos os atores de uma simulação que se encontram dispersos pelos vários *Local Coordinators*.

Para a implementação de mensagens *Broadcast* bastaria criar uma entrada na lista global de mensagens, dirigida às mensagens *Broadcast*, onde todos os atores a poderiam lá colocar e ler as mensagens *Broadcast*.

5.1.6.4 *Testes*

Tal como nas outras soluções foi então realizada uma monitorização ao comportamento do seu protótipo durante a execução. Foi feita uma simulação com uma duração de 30 minutos e 20.000 atores, a enviarem uma mensagem e a receberem as mensagens destinadas a si a cada intervalo de 500 milissegundos. Nestas condições a situação ideal seria serem geradas 2.400.000 mensagens após 1 minuto, 24.000.000 mensagens após 10 minutos e 72.000.000 mensagens após 30 minutos. Foi usado um computador com um processador i7 3610QM, 6Gb de memória ram e com o sistema operativo Windows 8 64 bits. Os resultados desta simulação foram os seguintes:

	CPU	Memória	Mensagens Geradas
1 minuto	5,7%	1509,0 Mb	2.376.971
10 minutos	5,7%	1025,9 Mb	23.931.651
30 minutos	6,0%	914,6 Mb	71.880.697

Tabela 9: Envio de Mensagens: Solução 5 - Testes

Tal como na segunda e quarta solução, o número de mensagens geradas é bastante próximo do ideal e também com tendencia a aproximar-se.

A carga de CPU é também estável e baixa durante toda a simulação.

5.1.7 *Conclusões*

Após efetuados os testes aos protótipos de cada solução, pode-se então concluir qual a solução que responderá de melhor forma às necessidades do simulador.

Na Figura 45 encontra-se um gráfico com os valores retirados dos testes às simulações ao instante de simulação 30 minutos, pois embora não existam variações muito significativas durante os testes, o instante de tempo 30 minutos é o que possivelmente mais se aproximará do processamento normal sem interferências de atrasos na criação inicial de *threads*.

Está então representada a vermelho a percentagem da carga de CPU, e a azul a percentagem de mensagens geradas relativamente ao valor ideal de mensagens geradas, significando assim que idealmente os valores a vermelho deverão ser o mais próximo possível de 0%, e os valores a azul o mais próximo possível de 100%.

À partida foi facilmente descartada a hipótese do uso da solução 1, pois esta apresenta uma eficiência muito baixa relativamente à geração de mensagens.

A solução 3 foi também facilmente descartada pois apresenta uma carga de CPU muito elevada, e demostra também uma baixa eficiência na geração de mensagens.

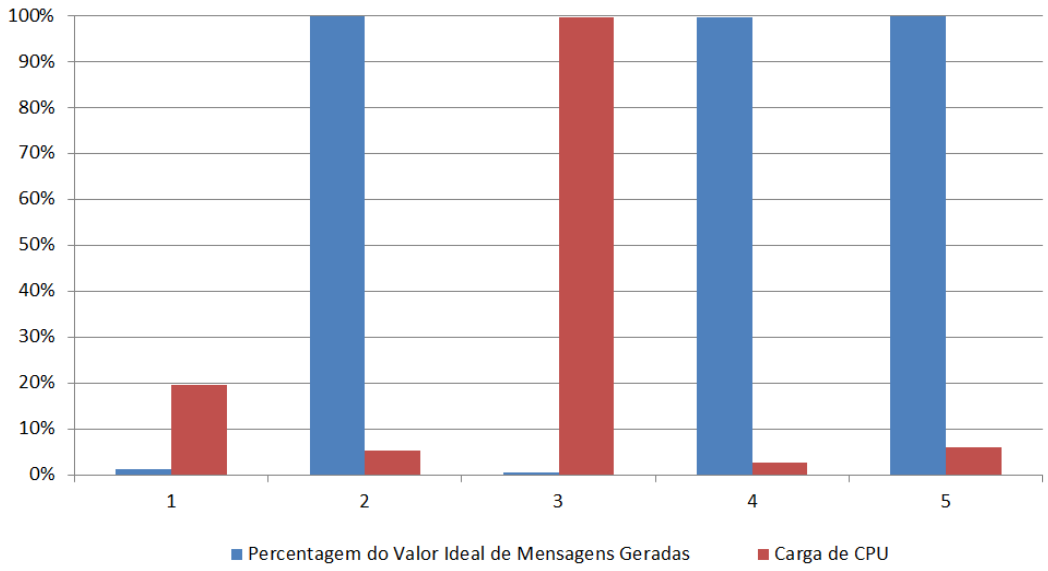


Figura 45: Comparação das soluções

Os testes realizados às soluções 2, 4 e 5, apresentam resultados bastante semelhantes, quer em CPU, memória ou mensagens geradas e todos bastante positivos com uma alta eficiência na geração de mensagens, e baixo consumo de CPU. Desta forma será necessário optar pela solução que melhor se enquadrará no simulador.

Tal como já foi explicado, visto o simulador estar desenhado e construído de forma distribuída, a implementação da solução 4 obrigaria o uso de técnicas RDMA que levaria um aumento da complexidade do simulador, e uma perda no controlo das mensagens enviadas e recebidas entre diferentes *Local Coordinators*. Da mesma forma, esta solução torna-se incapaz de responder de forma direta ás mensagens de *Broadcast*, levando a estas fossem convertidas em mensagens *Unicast* que aumentaria a carga na rede. Esta solução foi assim descartada, não tendo sido em conta como uma solução viável.

Ficando assim a escolha limitada pelas duas restantes soluções, 2 e 5, foi então necessário optar pela que pareça responder melhor aos requisitos do simulador. Após uma longa análise a ambas soluções, foi escolhida a solução 2 como a que melhor responderá aos requisitos deste simulador, por duas razões. A primeira razão, é que os testes aos protótipos destas soluções apresentaram resultados muito idênticos, contudo indiciando uma pequena vantagem na performance da solução 2, com uma ligeira diminuição na carga de CPU e consumo de memória. A outra razão é pelo

facto de o simulador estar desenhado e construído de forma distribuída, implicando assim que na solução 5 haja uma classe gestora de mensagens em cada máquina onde estejam a correr atores, o que iria aumentar ainda mais a complexidade desta solução perante a solução 2, sem que haja tal necessidade pois o método de escrita e leitura das mensagens da lista global não apresenta um grande nível de complexidade ao ponto de ser necessário separar essa parte da classe ator.

5.2 BARTUM - DISTRIBUIÇÃO DE MENSAGENS

5.2.1 *Descrição do Problema*

Após escolhida a melhor solução para fazer o armazenamento local das mensagens, foi então necessário desenhar e implementar uma solução para a distribuição destas entre os vários elementos do simulador (*Local Coordinators*, *Global Coordinator* e *Vizualization*). Tal torna-se essencial, pois como este simulador é distribuído e contém assim os atores dispersos por diferentes *Local Coordinators*, tem de existir uma troca de mensagens entre os mesmos, para que os atores possam comunicar entre si. É também necessário que as mensagens passem pelo *Global Coordinator* para que este as tenha em atenção para fazer o *reporting* das comunicações. O *Vizualization* tem também de receber as mensagens que passam na rede, para que possa apresentar informação ao utilizador sobre estas.

Foi então necessário chegar a uma solução que satisfizesse todos os requisitos do sistema, e respeitasse regras de fiabilidade tais como garantir o envio e recepção de todas as mensagens, não permitir mensagens duplicadas ou ainda fazer o tratamento adequado para as mensagens de *Broadcast*.

5.2.2 *Solução*

A primeira ideia encontrada para solucionar este problema, passava pela simples sincronização da lista de mensagens partilhada, pelos diferentes elementos do simulador (*Local Coordinators*, *Global Coordinator* e *Vizualization*), ideia esta baseada na sincronização da lista de estados dos atores. Para tal, seria necessário criar uma ligação *multicast* entre todos, e fazer a partilha de todas as mensagens de cada lista de mensagens.

Contudo, face a requisitos técnicos, tais como a não duplicação de mensagens ou o correto envio de mensagens *broadcast*, esta solução não serve pois a simples sincronização das listas de mensagens não permitiria controlar devidamente estes requisitos do simulador.

Foi então pensada, desenhada e elaborada, uma solução mais robusta, que permitisse fazer o controlo adequado de todas as mensagens e disseminação destas, por todos os elementos do simulador. A Figura 46 mostra o desenho inicial desta solução.

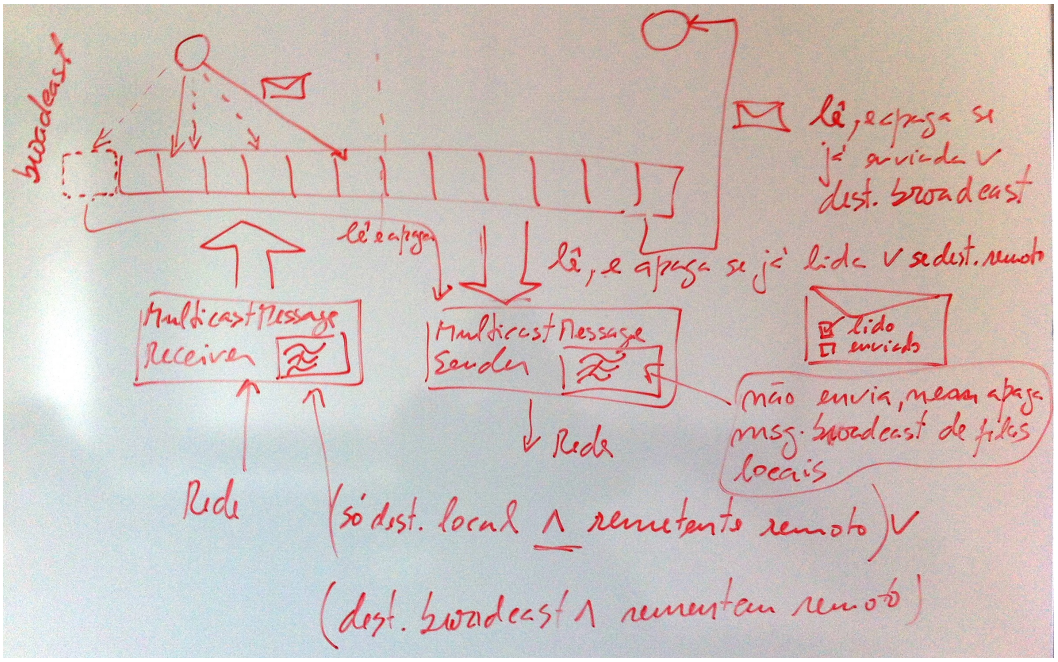


Figura 46: Desenho da Solução

Como se pode ver pela Figura 46, foram criadas duas entidades responsáveis pelo envio e recepção das mensagens, sendo estas a *Multicast Message Sender* e a *Multicast Message Receiver*. Estas entidades são então responsáveis pelo controlo do envio e recepção de todas as mensagens, sendo elas as responsáveis por as tirar/pôr na lista global de mensagens, e receber/enviar para a rede.

Tal como o nome indica, estas entidades criam uma ligação *multicast* entre os diferentes elementos do simulador, e para tal é lido do ficheiro de configurações o endereço *multicast* e a porta, para que todos consigam comunicar.

O uso de uma ligação *multicast*, vem trazer um problema de fiabilidade na recepção de mensagens, visto que o *multicast* usa o protocolo UDP que não oferece garantias na entrega de pacotes. Contudo, após pensado, discutido e testado, foi decidido usar *multicast* pois como se trata de uma rede local, muito dificilmente ocorrem perdas de pacotes, tal como foi evidenciado nos testes feitos, em que a perda de pacotes foi nula nas várias simulações efetuadas. Assim, com o uso do *multicast* podemos tirar vantagem do rápido envio de mensagens, entre todos os elementos da rede.

5.2.2.1 *Classe Message*

Para representar uma mensagem de dados ao nível do servidor, isto é, um objeto que identifica uma mensagem, o qual será armazenado na lista de mensagens e enviado/recebido via rede entre os elementos que compõem o simulador, foi implementada uma classe *Message*, tal como nos protótipos elaborados para o armazenamento de mensagens.

Esta classe armazena assim um conjunto de dados relativos à mensagem incluído campos de apoio à transferência de mensagens tal como o ID do ator origem e destino ou o instante de tempo em que a mensagem foi gerada. Nela circula também os dados da mensagem correspondendo à tecnologia de comunicações em questão. A Tabela 10 tem a listagem dos campos que circulam nesta mensagem e o tipos correspondentes.

Campo	Tipo
time_stamp	long
source_id	String
protocol	Char
actor_x	Double
actor_y	Double
data	String
read	Boolean
sent	Boolean

Tabela 10: Campos da Classe Message

O campo *time_stamp* guarda o instante de tempo em que a mensagem foi gerada. O campo *source_id* guarda o ID do ator que gerou a mensagem

e os campo *actor_x e actor_y* a posição deste no instante em que enviou a mensagem. O campo *data* guarda os dados da mensagem correspondente à tecnologia em questão que é representada pelo campo *protocol*. Os campos *read* e *sent* serão explicados mais à frente neste capitulo.

5.2.2.2 *Controlo do Envio e Leitura das Mensagens*

Como todas as mensagens, quer sejam destinadas a atores locais, atores remotos ou de *broadcast*, têm de ser enviadas para que possam chegar ao destinatário, ao *Global Coordinator* ou ao *Vizualization*, tem de existir um controlo para que estas não sejam apagadas, antes de terem sido lidas e enviadas. Para tal foram introduzidos dois atributos na classe *Message*, para que este controlo possa ser feito, e permitir assim um correto funcionamento no envio das mensagens.

O primeiro atributo adicionado foi uma *flag* de controlo de leitura. Foi então definida uma variável do tipo *boolean* que identifica se uma mensagem já foi lida ou não pelo destinatário, estando a *true* se sim, ou a *false* se não. Esta *flag* permite então que uma mesma mensagem não seja lida mais do que uma vez pelo destinatário, o que significaria uma duplicação da mensagem. Tal poderia acontecer, pois como todas as mensagens têm de ser enviadas para os outros elementos do simulador, o ator poderia procurar por mensagens mais rapidamente do que estas são despachadas pelo *Multicast Message Sender*.

O segundo atributo adicionado foi uma *flag* de controlo de envio da mensagem. Foi definida uma variável do tipo *boolean* que identifica se uma mensagem já foi enviada ou não, estando a *true* se sim, ou a *false* se não. Esta *flag* permite dessa forma que uma mensagem não seja disseminada na rede mais do que uma vez, que resultaria na mesma situação de mensagens duplicadas.

Com o uso destas duas flags, tona-se possível controlar a leitura e envio único das mensagens, permitindo também apagar as mensagens garantindo que estas já foram lidas e enviadas.

5.2.2.3 *Mensagens de Broadcast*

Para permitir o envio de mensagens de *broadcast* entre os atores, foi também necessário pensar numa forma de as integrar nesta solução, tornando assim possível fazer a gestão destas, tanto no que respeita ao envio/recepção por parte dos atores, bem como procurar fazer uma ocupação mais eficiente da rede, evitando tráfego desnecessário.

Sendo a lista de mensagens distribuída um *HashMap* com chave ID do ator, é criada uma entrada para um ID *Broadcast*. Assim, sempre que um ator pretende enviar uma mensagem *broadcast*, basta que para isso a coloque na posição com chave *broadcast* do *HashMap*. No entanto, a mensagem é também colocada em cada posição do *HashMap* com ID de um ator local. Tal é necessário, pois se para a leitura, apenas se usasse a lista de mensagens *broadcast*, seria complexo apagar estas mensagens e controlar a leitura destas, pois todos os atores estariam a aceder á mesma mensagem.

O *Multicast Message Sender*, tratará assim de enviar as mensagens colocadas na posição *broadcast* do *HashMap*, para todos os outros elementos do simulador. Estes porém, com o uso do *Multicast Message Receiver*, ao receberem uma mensagem *broadcast*, tratam de a colocar em todas as posições do *HashMap* com chave de atores locais.

Esta técnica permite assim que uma mensagem *broadcast* enviada, corresponda apenas a uma mensagem na rede, não saturando esta com inúmeras mensagens destinadas a todos os atores presentes na simulação.

A Figura 47 exemplifica o processo de envio de uma mensagem *broadcast*. Neste caso, o ator local pretende enviar uma mensagem *broadcast*, e dessa forma coloca a mensagem na lista de mensagens de cada ator local, e também na lista de mensagens *broadcast*, para que esta seja disseminada pelos outros elementos do simulador.

Já na Figura 48, podemos ver o processo oposto, a recepção de mensagens *broadcast*. Quando chega uma mensagem *broadcast* pela rede, o *Multicast Message Receiver*, é então responsável por a colocar nas listas de mensagens, pertencentes a todos os atores locais.

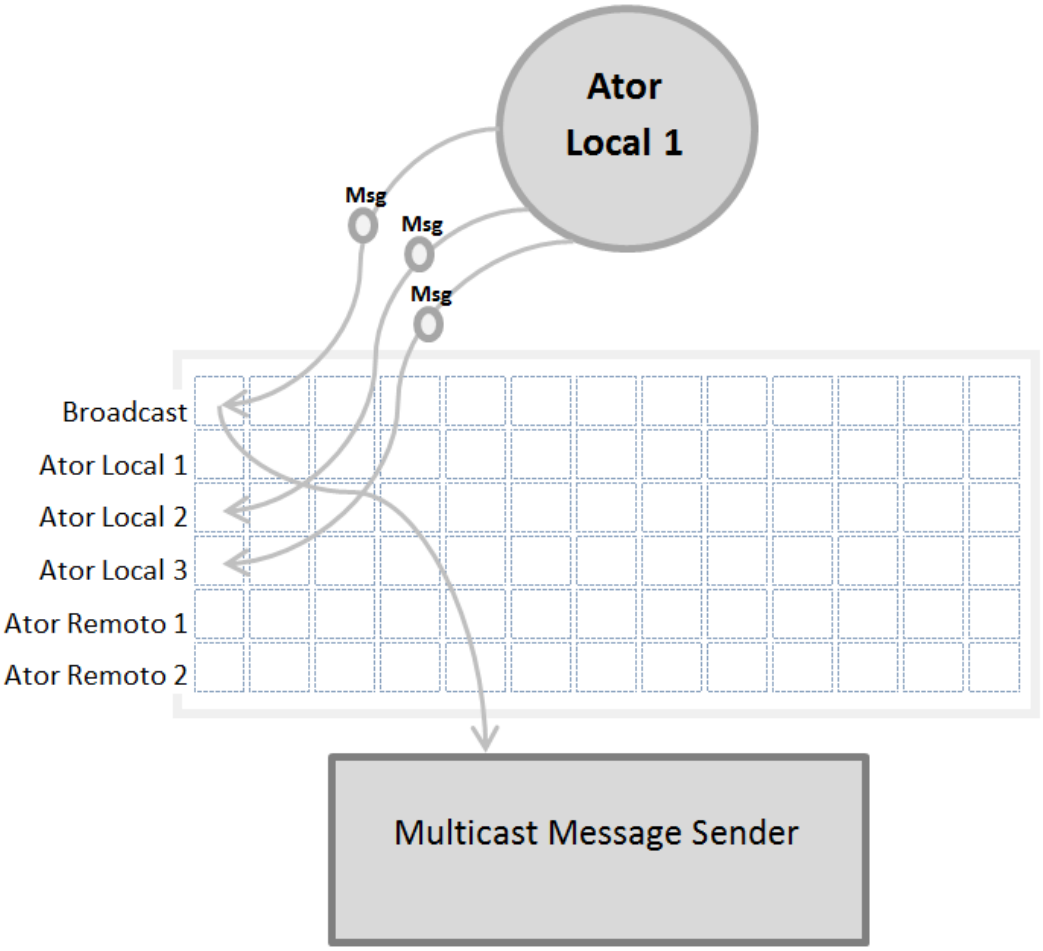


Figura 47: Envio de Mensagens Broadcast

5.2.2.4 *Multicast Message Sender*

Foi então criada uma classe *MulticastMessageSender* no projeto, para fazer o controlo do envio das mensagens para todos os elementos do simulador. Esta classe cria uma *thread* que periodicamente percorre toda a lista de mensagens e as envia para a rede de forma controlada, respeitando várias regras definidas no desenho desta solução, das quais algumas já foram descritas. A Figura 49 mostra um diagrama funcional desta classe.

Percorrendo todas as listas de mensagens de cada ator, uma a uma esta classe começa por verificar se se trata da lista de mensagens *broadcast* ou de uma lista de mensagens de um ator remoto. Se for, percorre todas as mensagens, marca-as como enviadas e envia-as via *multicast*.

No caso de a lista de mensagens ser a lista de um ator local, são percorridas todas as mensagens, e caso estas não sejam de *broadcast*, envia e apaga caso já tenham sido lidas, ou apenas envia caso ainda não tenham sido

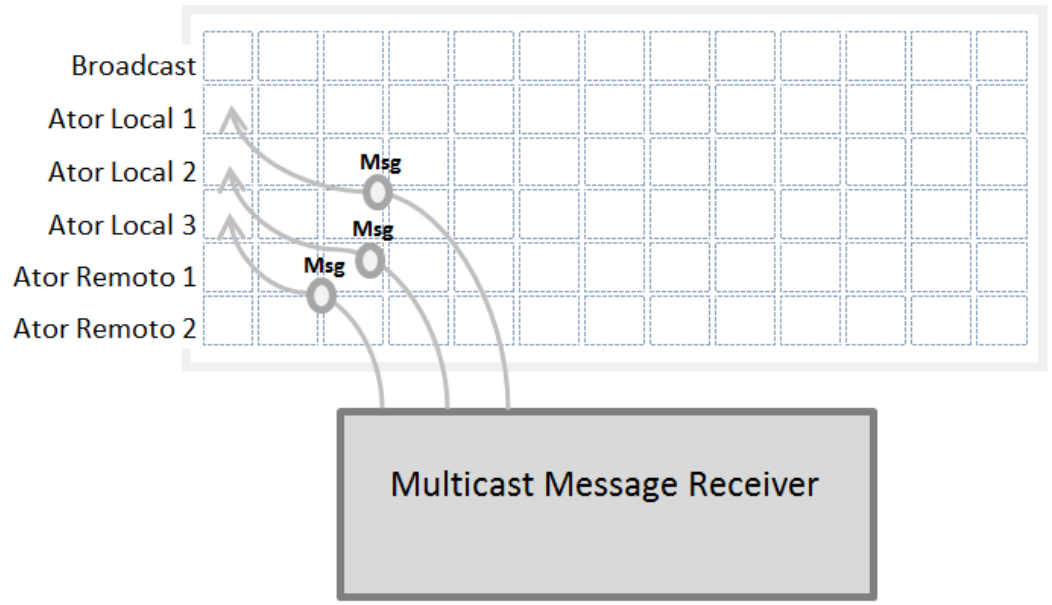


Figura 48: Recepção de Mensagens Broadcast

lidas. Se forem de *broadcast*, estas são ignoradas, pois são mensagens já recebidas, as mensagens *broadcast* a enviar, são colocadas na lista de mensagens *Broadcast*.

Para uma melhor perceção, a Figura 50 apresenta um fluxograma deste mecanismo.

ENVIO DE MENSAGENS

Numa procura de otimização, o envio de mensagens por parte do *Multicast Message Sender*, foi implementado de forma a enviar múltiplas mensagens num único pacote. Para tal, quando é pretendido enviar uma mensagem, verifica-se primeiro se esta cabe no pacote, se sim é incrementada ao pacote e enviada no fim do processo de envio de mensagens, ou quando o pacote estiver cheio. Caso a mensagem não caiba no pacote, é enviado o pacote de mensagens, e gerado um novo com esta nova mensagem. O tamanho máximo de um pacote foi definido a 1500 bytes, que é o tamanho máximo de um pacote UDP para que não seja fragmentado.

Para fazer a junção das diversas mensagens num mesmo pacote, foi necessário desenhar uma solução para as compactar e descompactar, visto estas poderem ter tamanhos distintos. Para tal, em cada pacote é envi-

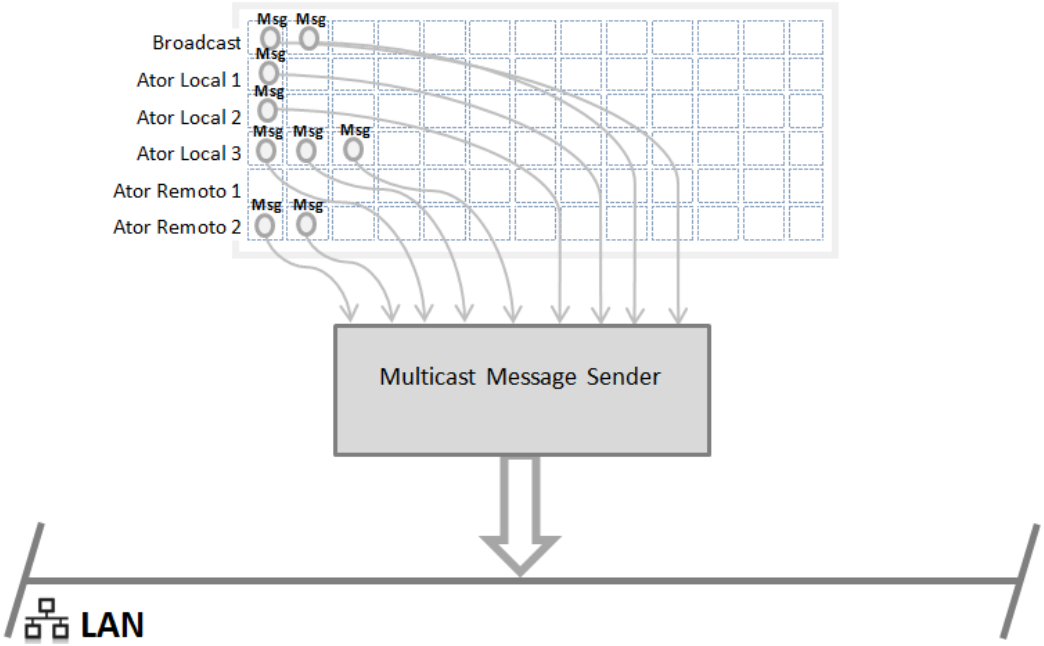


Figura 49: Multicast Message Sender

ado o número de mensagens presentes neste, e antes de cada mensagem é enviado o tamanho desta. Assim torna-se possível separar as mensagens, sabendo exatamente onde começa e termina uma mensagem.

5.2.2.5 *Multicast Message Receiver*

Da mesma forma, foi criada uma classe *MulticastMessageReceiver* no projeto, para fazer o controlo da recepção das mensagens provenientes de outros *Local Coordinators*. Esta classe cria uma *thread* que periodicamente faz uma leitura por pacotes recebidos na rede, e as coloca nas devidas listas de mensagens. A Figura 51 mostra um diagrama funcional desta classe.

Visto que um pacote enviado por um *Local Coordinator*, pode conter várias mensagens, para cada pacote recebido são retiradas todas as mensagens. Para cada mensagem, é visto se esta é uma mensagem *broadcast*, ou uma mensagem endereçada a um ator local. No caso de ser uma mensagem *broadcast*, a mensagem é colocada na lista de mensagens de todos os atores locais. No caso de ser uma mensagem destinada a um ator local, esta é unicamente colocada na lista de mensagens do ator em questão.

Para uma melhor perceção, a Figura 52 apresenta um fluxograma deste mecanismo.

5.2.3 *Message Reporting*

Para análise das comunicações da simulação, durante esta são registados todos os eventos de mensagens enviadas, recebidas e descartadas pelos diversos *Local Coordinators*, sendo assim possível fazer a posterior análise das comunicações de uma simulação.

Para tal foi criada uma classe *MessageReporting* para tratar da gestão de armazenamento das mensagens no ficheiro de reporting. Este classe tem assim três métodos para fazer esta gestão, o método *reportSendMessage* que escreve para o ficheiro de reporting o registo das mensagens enviadas. O método *reportReceivedMessage* que escreve para o ficheiro de reporting o registo das mensagens recebidas, e o método *reportDropMessage* que escreve para o ficheiro de reporting todos os registos de mensagens descartadas por estarem fora de alcance no momento de envio.

Foi então pensado e analisado qual seria o melhor formato para fazer o registo de mensagens enviadas, recebidas e descartas. Foi então decidido adotar o mesmo formato dos ficheiros de reporting do simulador de redes NS2. São usadas tags que identificam os dados, os quais poderão ser: tipo de registo, instante de tempo, origem, destino, posição e motivo de descarte de um pacote.

O Listagem 10 mostra um excerto de um ficheiro de reporting de mensagens. A primeira letra representa o tipo de registo onde 'r' representa uma mensagem recebida, 's' representa uma mensagem enviada e 'd' representa uma mensagem descartada.

Listagem 10: Ficheiro de Message Reporting

```
1 r -t 1379760153423 -Ni Ped1.2 -Nx -840284.0875632542 -Ny
     4156800.2095012246 -Hs Ped1.2 -Hd Ped1.0
2 r -t 1379760153428 -Ni Ped1.1 -Nx -840103.9084932259 -Ny
     4156896.879464813 -Hs Ped1.1 -Hd Ped1.0
```

```
3 s -t 1379760158332 -Ni Ped1.0 -Nx -840093.316756485 -Ny
    4156961.617038899 -Hs Ped1.0 -Hd Ped1.2
4 s -t 1379760158333 -Ni Ped1.0 -Nx -840093.316756485 -Ny
    4156961.617038899 -Hs Ped1.0 -Hd Ped1.2
5 d -t 1379760158332 -Ni Ped1.0 -Nx -840093.316756485 -Ny
    4156961.617038899 -Nw Out_Of_Range(256.07000883142723) -Hs Ped1.0 -
    Hd Ped1.2
6 d -t 1379760158333 -Ni Ped1.0 -Nx -840093.316756485 -Ny
    4156961.617038899 -Nw Out_Of_Range(256.07000883142723) -Hs Ped1.0 -
    Hd Ped1.2
7 s -t 1379760159434 -Ni Ped1.0 -Nx -840082.5352041859 -Ny
    4156966.0954841403 -Hs Ped1.0 -Hd Broadcast
8 s -t 1379760159434 -Ni Ped1.0 -Nx -840082.5352041859 -Ny
    4156966.0954841403 -Hs Ped1.0 -Hd Broadcast
9 s -t 1379760159434 -Ni Ped1.0 -Nx -840082.5352041859 -Ny
    4156966.0954841403 -Hs Ped1.0 -Hd Broadcast
10 s -t 1379760159434 -Ni Ped1.0 -Nx -840082.5352041859 -Ny
    4156966.0954841403 -Hs Ped1.0 -Hd Broadcast
```

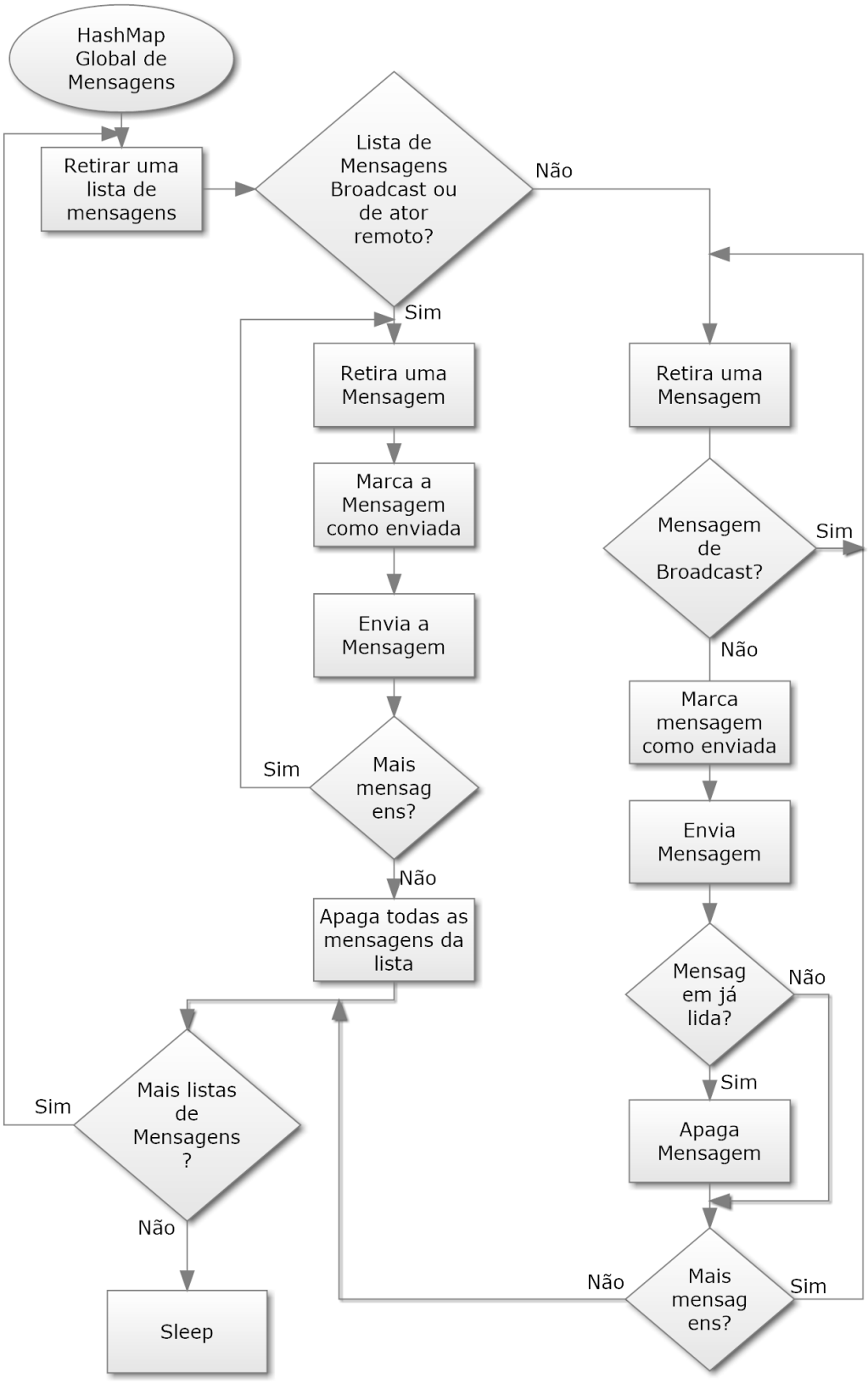


Figura 50: Multicast Message Sender: Fluxograma

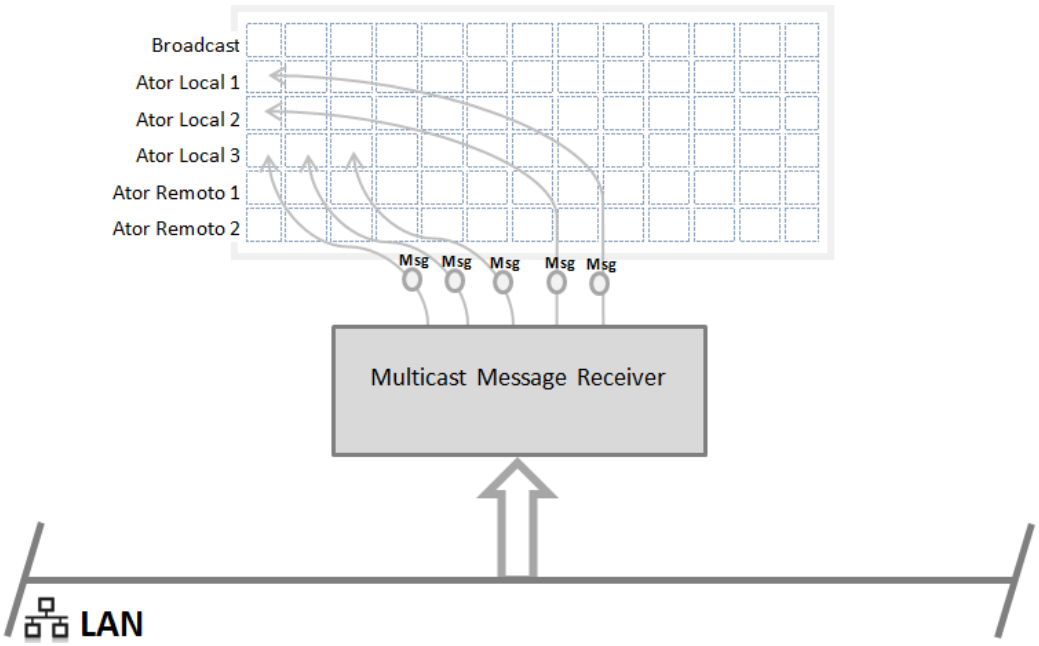


Figura 51: Multicast Message Receiver

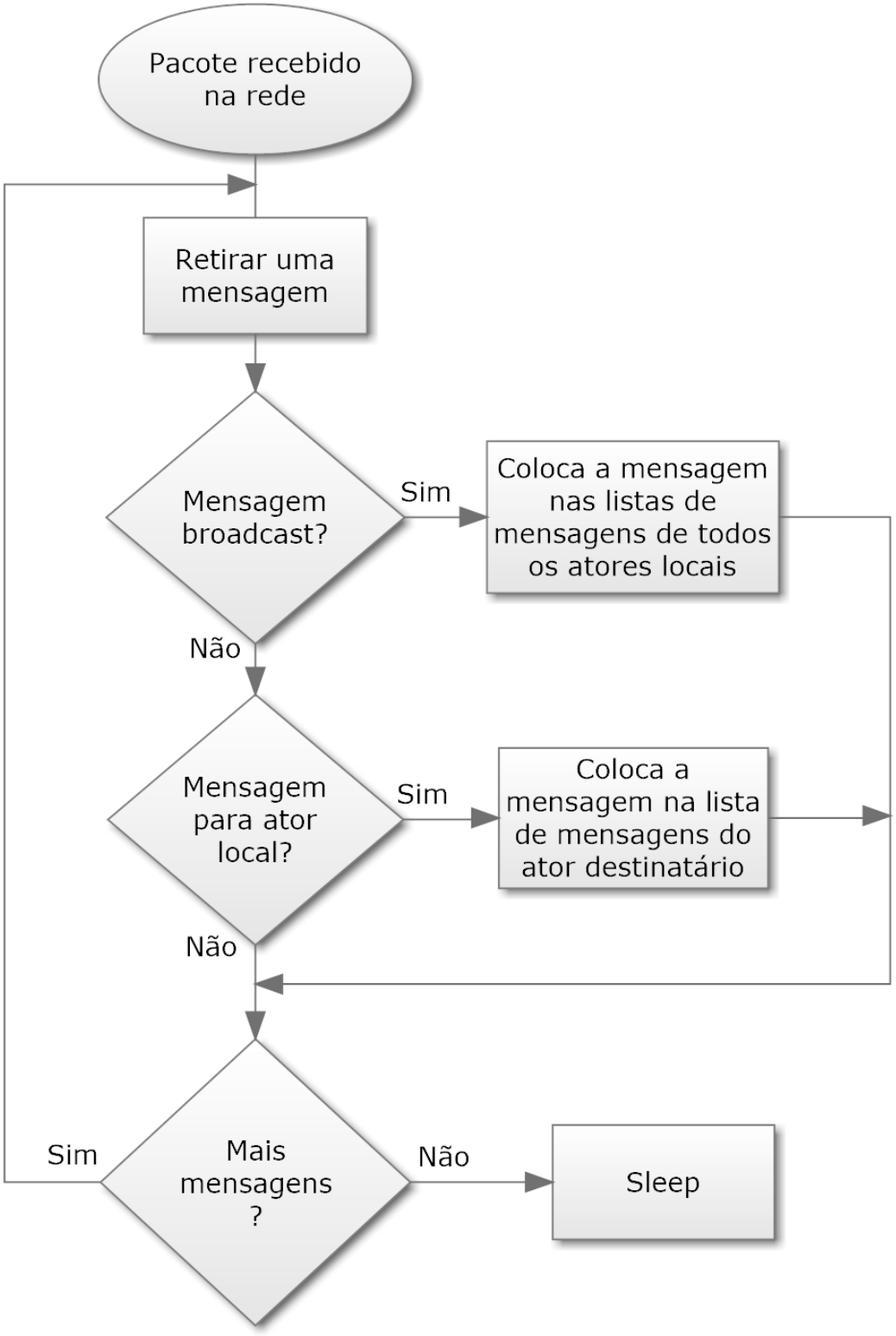


Figura 52: Multicast Message Receiver: Fluxograma

6

BARTUM - SIMULAÇÃO BLUETOOTH

Após encontrada e implementada a solução para a distribuição das mensagens, torna-se então possível ter um sistema de envio de mensagens entre atores, quer estes estejam no mesmo local coordinator ou em local coordinators separados. Desta forma, e sendo objetivo desta dissertação foi então pensada, desenhada e implementada uma pilha protocolar para para fazer a simulação da tecnologia Bluetooth.

Com a análise da tecnologia Bluetooth foram analisadas as camadas da pilha protocolar com mais relevância, as quais foram desenhadas e implementadas. As secções seguintes explicam como foi implementada esta pilha protocolar.

6.1 BLUETOOTH CHANNEL

Foi implementada uma classe *BluetoothChannel*, responsável por fazer toda a gestão das mensagens entre a interface Bluetooth e o modelo de dados criado para o armazenamento das mensagens, sendo então responsável por ler e colocar na estrutura de mensagens, todas as mensagens trocadas pelo interface Bluetooth.

Para fazer o controlo destas mensagens, foi criado um atributo *protocol* do tipo *char* na classe *Message*, para que as mensagens no simulador possam ser distinguidas por protocolo, garantindo assim que as interfaces apenas mexem nas mensagens relativas à interface em questão. Neste caso, as mensagens relativas ao protocolo Bluetooth têm como *protocol* o *char* 'b'.

Foram implementados dois métodos base nesta classe, o método *receiveMessages* que tal como o nome indica trata da leitura das mensagens Bluetooth da estrutura de mensagens, e o método *sendMessage* que coloca as mensagens de envio na estrutura de dados das mensagens.

6.1.1 *sendMessage*

O método *sendMessage* é sempre evocado pela camada imediatamente acima implementada, e recebe como parâmetro um *BitSet* correspondendo a uma frame de dados da camada física do protocolo Bluetooth. Este método começa por gerar uma mensagem do tipo Message, preenchendo os seus atributos como o instante de tempo em que a mensagem foi gerada, o ator origem, ator destino, o tipo de protocolo da mensagem, posição do ator e os dados recebidos por parâmetro. De seguida verifica se se trata de uma mensagem *Broadcast* ou de uma mensagem *Unicast*. Caso esta seja Broadcast este coloca-a na posição *Broadcast* do *HashMap* de mensagens, caso contrário coloca-a na posição do ator destino, para que estas sejam depois enviadas para a rede, e lidas pelo destinatário. A Figura 53 mostra um fluxograma deste método.

6.1.2 *receiveMessages*

O método *receiveMessages* ao contrário do método *sendMessage*, é o responsável pela leitura das mensagens *Bluetooth* que se encontram no *HashMap* de mensagens, e também responsável de as passar para a camada imediatamente acima implementada. Este método começa assim por bloquear o acesso à lista de mensagens do ator no HashMap, de seguida percorre todas as mensagens da lista e uma a uma verifica se esta é uma mensagem da interface *Bluetooth*. Caso não seja, não faz nada e passa à mensagem seguinte, mas se esta for uma mensagem correspondente à interface *Bluetooth*, este verifica se a mensagem estaria no alcance *Bluetooth* e seria assim eventualmente recebida, para a passar à camada imediatamente acima ou não. A Figura 54 mostra um fluxograma deste método. Para verificar a receção da mensagem, foi implementado o método *verifyMessageReception*.

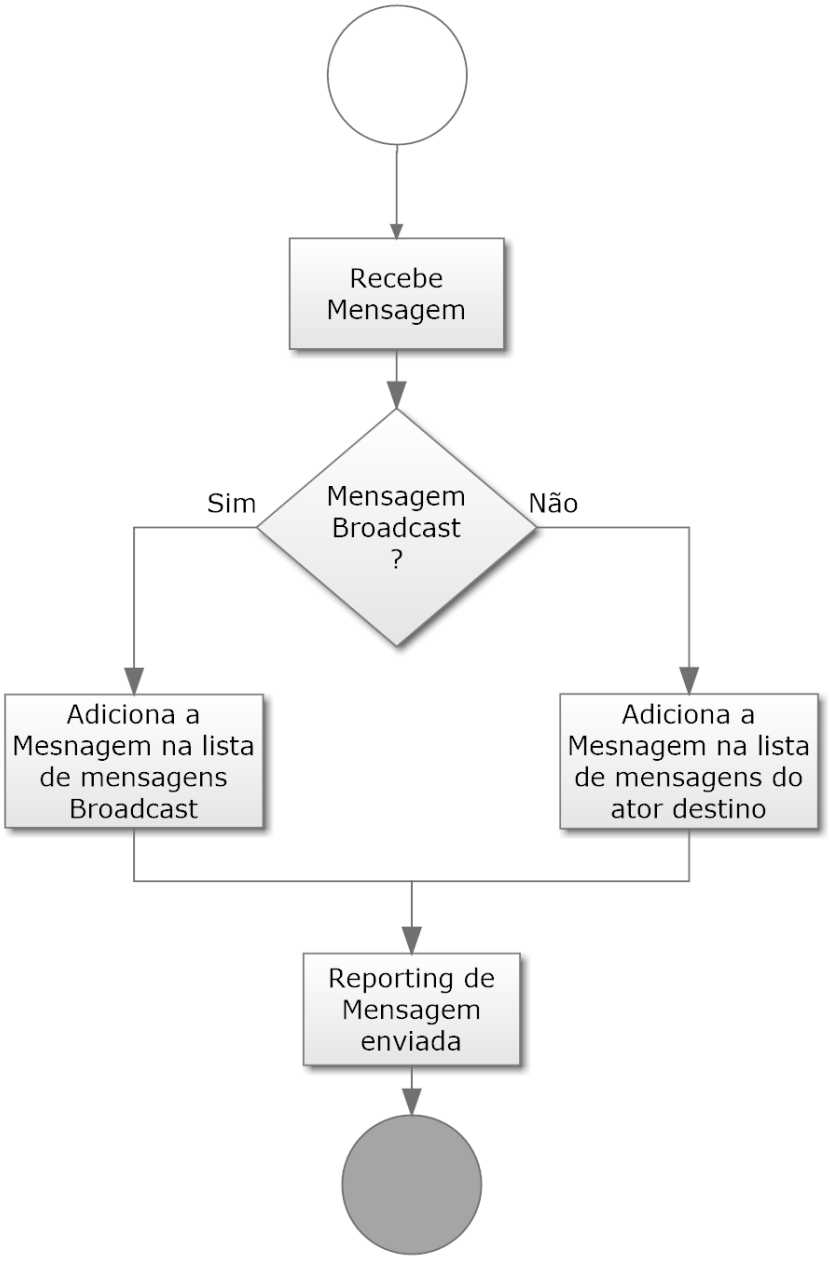


Figura 53: SendMessage Fluxograma

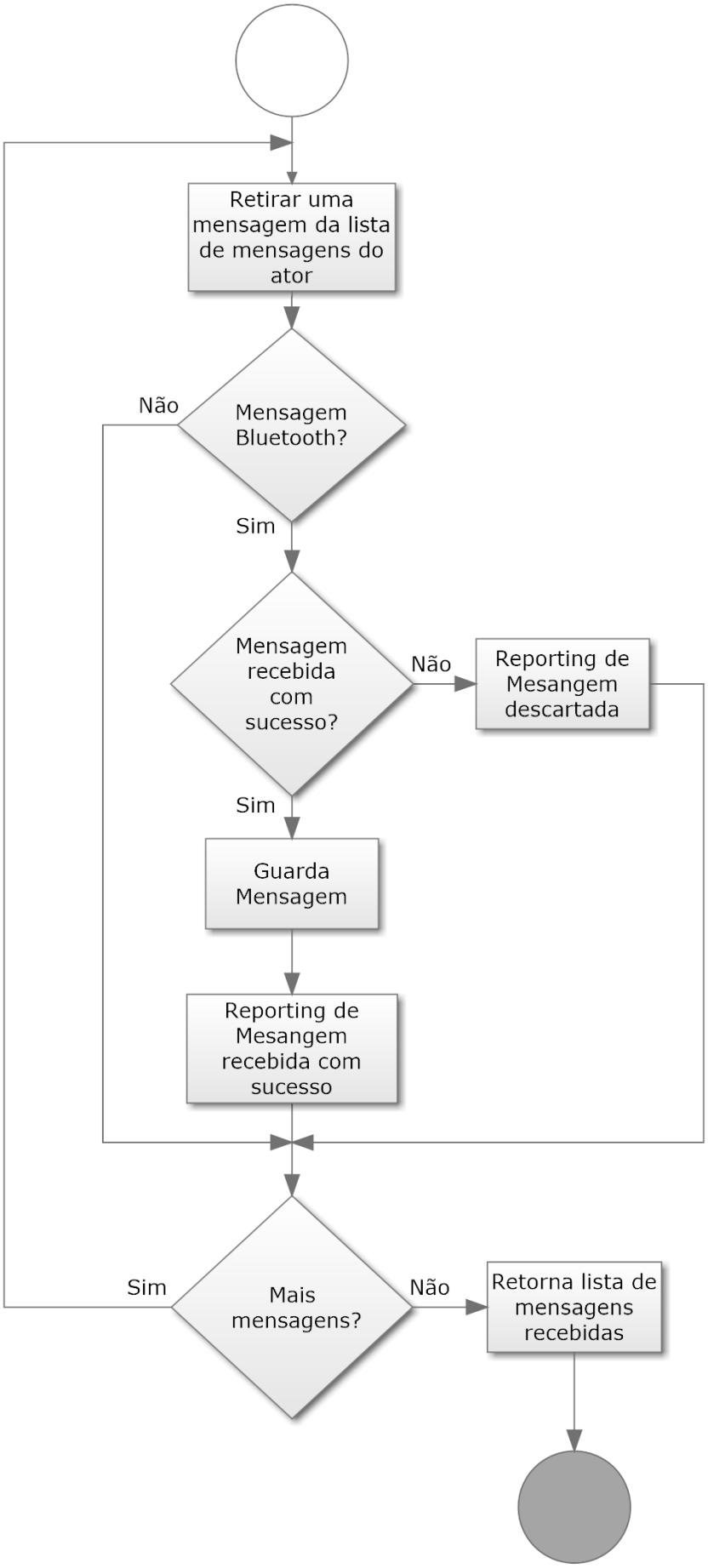


Figura 54: ReceiveMessages Fluxograma

6.1.3 *VerifyMessageReception*

Este método tem como funcionalidade verificar se uma mensagem *Bluetooth* recebida na lista de mensagens, teria sido realmente recebida tendo em conta o alcance da tecnologia, verificando assim se o dispositivo emissor estaria no alcance do dispositivo receptor no momento do envio da mensagem.

Para fazer esta verificação foi implementada uma formula capaz de calcular a distância entre duas coordenadas geográficas, verificando se estas estariam distanciadas a menos de um determinado valor definido em metros, no caso do *Bluetooth* definida em 10 metros. A Figura 55 mostra esta equação.

$$Distancia = ((ArcoCosseno(Cosseno((90 - LatitudeUm) * \pi/180) * Cosseno((90 - LatitudeDois) * \pi/180) + Seno((90 - LatitudeUm) * \pi/180) * Seno((90 - LatitudeDois) * \pi/180) * Cosseno(ABS(((360+LongitudeUm)*\pi/180)-((360+LongitudeDois)*\pi/180))))) * 6371,004) * 1000$$

Figura 55: Equação Distância Geográfica

Para realizar este cálculo, têm de ser usadas as coordenadas de ambos dispositivos no instante de tempo em que a mensagem foi enviada. Mas como este simulador tem uma arquitetura distribuída, e dessa forma as mensagens poderão não ser recebidas imediatamente após terem sido enviadas, foi criado um histórico de posições *position_history* na classe ator, permitindo assim que sejam guardados um determinado número de anteriores posições do ator na simulação.

Assim quando este método é chamado, é procurada nessa lista de posições anteriores o instante de tempo mais próximo, para que sejam comparadas as posições de ambos dispositivos para o "mesmo"instante de tempo. É também tido em conta que esse intervalo de tempo não seja superior a 100 milissegundos, pois sendo que um ator atualiza a sua posição com esta frequência de tempo, este seria o máximo de tempo necessário para o caso de estes estarem com a sua frequência de atualização totalmente dessincronizada.

6.1.4 *Discover Devices*

Esta funcionalidade tal como o nome indica tem como fim a procura de dispositivos vizinhos. Para tal cria uma ligação física temporária entre os dispositivos, existindo apenas uma breve transação entre estes, não estando este serviço associado a nenhuma camada da pilha protocolar da tecnologia Bluetooth.

Tal como já foi explicado anteriormente, este serviço começa por enviar para o meio pequenos pacotes *ID Packet* de 68 bits com o *device access code* (DAC) ou o *inquiry access code* (IAC).

De seguida, caso os outros atores se encontrem a fazer leituras de pedidos, estes ao receberem um *ID Packet* poderão assim responder com um pacote FHS ao dispositivo que enviou o *ID Packet*. Este ao receber de volta o pacote FHS passa a saber assim quais são os dispositivos que estao próximos e aos quais pode estabelecer uma conexão, pois com o pacote FHS passa a ter todas as condições para estabelecer uma ligação com o dispositivo. A Figura 56 mostra o processo deste mecanismo.

Para implementar este serviço foi criada na classe BluetoothChannel um mecanismo que envia uma mensagem de Broadcast com o *ID Packet* do dispositivo, sendo que na solução este campo é representado pelo ID do ator na simulação, por exemplo "Ped1.5". Visto ser esta classe que faz todo o controlo das mensagens recebidas e enviadas, e sendo estas então recebidas ou não consoante o cálculo da distância entre dispositivos, permite que apenas os dispositivos vizinhos recebam este pacote e respondam se for caso disso com o pacote FHS.

Este método limita-se apenas a enviar uma mensagem Broadcast, e a receber mensagens FHS, descobrindo dessa forma os dispositivos vizinhos e passando essa informação diretamente à camada de aplicação.

6.2 BLUETOOTH PHYSICAL LAYER

Esta classe representa a camada física do protocolo Bluetooth. Nesta camada seguem pacotes Bluetooth de tamanho variável entre 126 bits e 2870 bits, dependendo do tamanho de dados que nela seguem. Esta camada

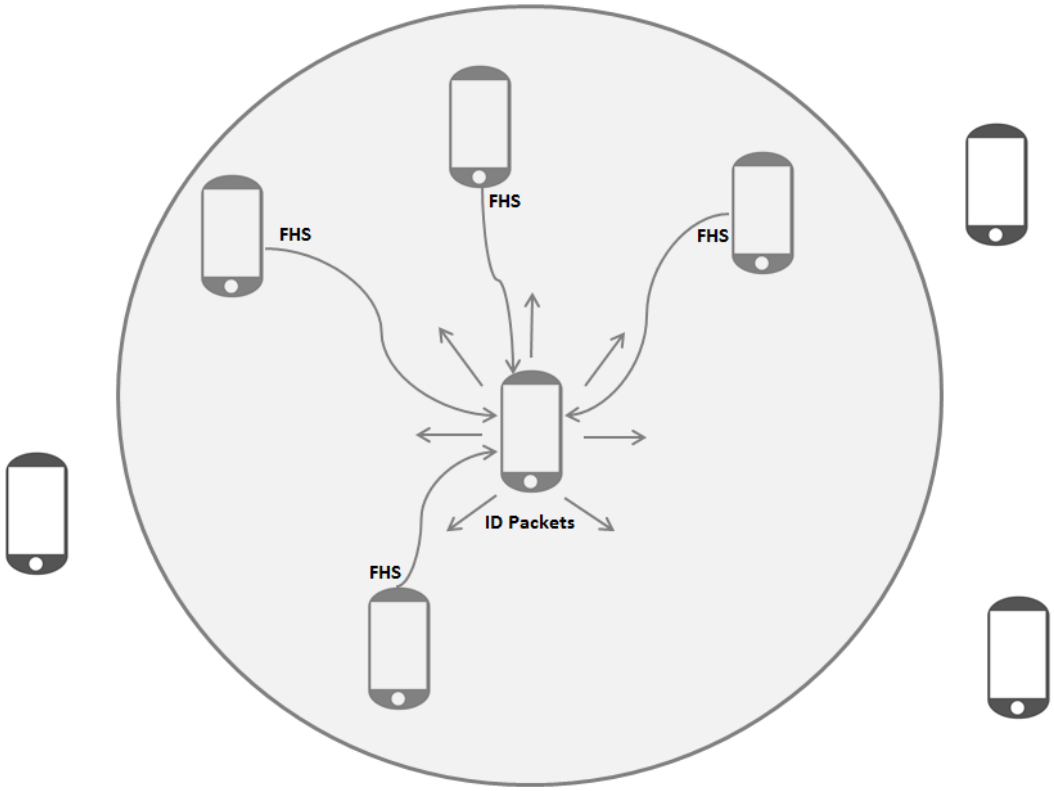


Figura 56: Discover Devices

tem tanto a função de enviar pacotes para o meio (*BluetoothChannel*) tal como receber pacotes vindos deste.

Foram então implementados dois métodos base nesta classe para fazer o controlo do envio e recepção dos pacotes, o método *sendPacket* e o método *receivePackets*.

6.2.1 *sendPacket*

O método *sendPacket* tal como o nome indica, é o método capaz de enviar um pacote Bluetooth para o meio. Este método recebe uma frame da camada superior (*BasebandLayer*) e constrói um pacote Bluetooth.

Um pacote Bluetooth é constituído por três partes: *Access Code*, *Header* e *Payload*. O *Access Code* tem um tamanho de 72 bits e é usado para identificar um pacote como sendo proveniente ou originário de um *Master*. O conteúdo do *Access Code* foi ignorado visto não ser relevante para esta si-

mulação, sendo assim preenchido todo a zeros.

O *Header* tem o tamanho de 54 bits e é composto por um conjunto de campos de identificação e de controlo, tal como o identificativo do destinatário na *piconet/scatternet*, o tipo de pacote, flags de controlo de fluxo e controlo de erros. Para além de um campo *Header Error Check* que se trata de um simples CRC, é usado o mecanismo *Forward Error Correction* (FEC) que replica três vezes todos estes campos, permitindo assim uma possível correção de erros no *Header*.

Os campos de controlo de fluxo foram definidos todos a zero visto ter sido decidido ignorar o mecanismo de controlo de fluxo por dois motivos. O primeiro é a grande improbabilidade de ocorrerem perda de pacotes na rede, visto esta ser sempre uma rede local. O segundo motivo é a poupança de recursos, evitando assim que sejam enviados pacotes *ACK* como resposta a cada pacote recebido, o que iria praticamente duplicar a carga na rede.

O tipo de pacote foi definido a "0100"representando assim um pacote do tipo ACL. O controlo de erros também foi ignorado, e preenchido todo a zeros visto a grande improbabilidade de ocorrência de erros na rede, e à poupança de recursos de processamento.

6.2.2 *receivePackets*

O método *receivePackets* tal como o nome indica, é o método responsável por fazer a leitura de pacotes do meio. Este método limita-se apenas a chamar a função *receiveMessages* da camada imediatamente abaixo (*BluetoothChannel*), e a desmontar o pacote para passar à camada imediatamente acima, passando-lhe assim apenas o campo *payload* do pacote recebido.

6.3 BLUETOOTH BASEBAND LAYER

Tal como foi explicado na Seção 2.1, a camada *Baseband* da pilha protocolar do *Bluetooth* é a camada com a função de fazer a gestão de ligações entre dispositivos *Master* e *Slave*, sejam estas ligações ACL ou SCO.

Todas as ligações entre dispositivos neste simulador são consideradas ligações ACL, sendo estas as ligações usadas para troca de dados enquanto que as ligações SCO são usadas para para a comunicação com interfaces de voz que requerem mais rapidez na comunicação e sem tanta importância com controlo de perdas.

Quando a aplicação pretende procurar por dispositivos vizinhos, estes ao receberem um *ID packet* e ao responderem com um *FHS packet*, estão a fornecer todos os dados necessários para estabelecer uma conexão. Após isto torna-se possível estabelecer uma ligação. É então a aplicação que tem a possibilidade de fazer um pedido de conexão a outro dispositivo. É nesta camada que é gerado um pacote Bluetooth com um pedido de conexão na parte de dados do pacote. A camada Baseband do dispositivo alvo, ao receber o pedido de conexão, responde ao pedido e estabelece-se assim uma conexão ACL.

Foi criada nesta camada da pilha protocolar Bluetooth, uma lista de *Strings connections*, onde ficam guardados os IDs dos dispositivos, com quem existe conexão. Esta lista serve para fazer o controlo de conexões, sendo portanto necessário existir uma entrada nesta lista para que as aplicações de dois dispositivos possam comunicar entre si.

Desta forma, se uma aplicação tentar enviar uma mensagem para outro dispositivo sem ter estabelecido uma comunicação anteriormente, esta não passará pela camada Baseband, e será retornada com erro. Da mesma forma, se for recebida uma mensagem proveniente de um dispositivo para o qual já não existe uma conexão, esta mensagem também não será aceite.

Existem duas formas de se perder uma ligação, isto é, limpar a entrada ID do segundo dispositivo da lista *connections*. A primeira possibilidade passa pelo simples pedido de desconexão por parte da aplicação. Assim, se uma aplicação pretender terminar uma conexão, pode simplesmente fazer um pedido *sendDisconnectionRequest*, para que esta seja eliminada.

A outra possibilidade é que esta seja eliminada automaticamente, quando na camada *BluetoothChannel*, é recebido um pacote proveniente de um dispositivo ligado, e o qual é descartado por estar fora de alcance no momento em que este foi enviado, quebrando assim a ligação existente.

Esta camada da pilha protocolar Bluetooth, tal como todas as outras, tem a função de enviar e receber dados. É nesta camada que são montadas e desmontadas as frames ao nível da ligação, neste caso frames do tipo ACL. Existem assim dois métodos para tratar do envio e recepção de frames, o método *sendFrame* e o método *receiveFrames*.

6.3.1 *sendFrame*

O método *sendFrame* é então o método responsável pelo envio de frames para a camada física da pilha protocolar, este método recebe da camada imediatamente a cima (*L2CAP*) um BitSet com os dados a enviar.

Uma frame ACL tem um tamanho variável entre 32 bits e 2744 bits, dependendo do tamanho de dados enviados pela aplicação. Ao receber os dados da camada *L2CAP* este método encapsula-os em frames ACL.

Uma frame ACL é composta por 3 partes: *PayloadHeader*, *Payload* e *CRC*. O *PayloadHeader* contem informação relativa à frame, tal como o campo *Logical Channel* (L_CH) que diz se esta é uma continuação de outra frame ou o tamanho de dados que nela circulam. No *Payload* circulam os dados vindos da camada imediatamente a cima, e no CRC, tal como o nome indica o *Cyclic Redundancy Check* para fazer o controlo de erros, campo este ignorado e preenchido todo a zeros pela improbabilidade de ocorrência de erros na rede local do simulador, e para poupança de recursos de processamento.

6.3.2 *receiveFrames*

Este método ao contrário do método *sendFrame* é então o responsável por receber frames da camada imediatamente a baixo (*Baseband*) e desencapsular-los para a posterior leitura do destes pela camada imediatamente a cima (L2CAP). É então a camada *L2CAP* que chama este método no seu método de leitura.

6.4 BLUETOOTH L2CAP LAYER

Esta classe representa a camada *Logical Link Control and Application Protocol* (L2CAP) da pilha protocolar do Bluetooth. Tal como foi explicado na Seção 2.1, esta camada é responsável pela comunicação entre as várias possíveis camadas superiores de controlo de aplicações. Foi assim tida em conta como a camada de comunicação entre as aplicações e todo o resto da pilha protocolar.

Sendo esta a camada mais próxima da camada de aplicação, é então com esta camada que todas as aplicações comunicam. Para tal foram implementados dois métodos para fazer esta comunicação, o método *sendFrame* e o método *receiveFrames*.

6.4.1 *sendFrame*

Este método é então o responsável pelo envio de dados para o resto da pilha. Este limita-se assim a receber da aplicação uma *string* de dados, e a converte-la numa frame de bits para a passar à camada imediatamente a baixo, a camada *Baseband*.

6.4.2 *receiveFrames*

Mais uma vez o método *receiveFrames* é a responsável pela recepção de frames provenientes da camada imediatamente abaixo. E ao contrário do método sendFrame, esta recebe uma frame de bits, e converte-a numa string para passar à camada imediatamente acima, que são as aplicações.

6.5 BLUETOOTH APPLICATIONS

A camada de aplicação é a camada mais alta da pilha protocolar, sendo o ponto de contacto do utilizador com a tecnologia Bluetooth. Com todas outras camadas da pilha protocolar, as aplicações apenas têm de enviar e ler informação, sem se terem de preocupar com qualquer mecanismo de comunicação.

Após toda a pilha implementada, foram então criadas duas aplicações para testar a pilha protocolar, a aplicação AppPingPong e a aplicação AppServicesDiscoverProtocol.

6.5.1 *AppPingPong*

Esta aplicação tem a simples funcionalidade de enviar mensagens "ping"para um ator destino aleatório, e escutar por respostas. Para tal na classe da aplicação foi implementado um método *update* que é chamado pelo ator a cada atualização. Este método limita-se a ler e a apresentar as mensagens recebidas se estas existirem, e a gerar uma mensagem para um ator destino aleatório com uma probabilidade de uma em cada quinhentas vezes, evitando assim uma enorme afluência de tráfego na rede.

O método de envio de mensagem envia uma string com o texto "Ping", usando o método *sendFrame* da classe *BluetoothL2CAPLayer*. Por sua vez o método *receiveMessage*, usa o método *receiveFrames* da classe *BluetoothL2CAPLayer*, e verifica se para cada mensagem recebida, esta se tratava de uma mensagem "Ping", se for o caso, responde com uma mensagem "Pong".

6.5.2 *AppServicesDiscoverProtocol*

Outra aplicação implementada foi a *AppServicesDiscoverProtocol*. Esta aplicação simula o processo de descoberta de serviços do *Bluetooth*.

Nesta aplicação é pretendido que um dispositivo pesquise por serviços num dispositivo vizinho. Para tal foi implementado um método *update* que é chamado por todos os atores que possuem uma interface Bluetooth. Este método começa por verificar se se trata de um ator que pesquisa por dispositivos ou de um que apenas está a fazer leituras de pedidos.

No caso de ser um ator que pretende pesquisar serviços dos seus atores vizinhos, estas aplicação começa por chamar o método *dicoverDevices*. De seguida com a lista de todos os seus vizinhos, esta escolhe um vizinho aleatoriamente e faz-lhe um pedido de ligação. Com a ligação ACL estabelecida este pode então enviar-lhe um pedido de serviços. Por sua vez este ator que estará a fazer leitura de pedidos, poderá responder-lhe.

Neste caso, o ator que apenas está a fazer leituras de pedidos, antes de receber o pedido passa por todos os passos normais de conexão, tanto como resposta a *inquiries* para a descoberta de dispositivos tal como a reposta ao pedido de conexão ACL. E assim, com a conexão estabelecida e após recebido o pedido de serviços pode então enviar a lista dos seus serviços que será interpretada pela aplicação do outro dispositivo.

6.5.2.1 *Reporting*

Com vista a estudar o impacto da aplicação no simulador, e a obter resultados práticos desta, foi implementado um mecanismo de reporting da aplicação. Para tal foi implementada uma classe *AppDSPReporting* que tem a função de escrever para um ficheiro as ações da aplicação. São então registados as seguintes ações: enviado pedido de serviços disponíveis, enviada resposta de serviços disponíveis e recebida resposta de serviços disponíveis.

O ficheiro de reporting foi gerado em formato *Comma-separated values* (csv), formato este simples e de fácil leitura para que os dados possam ser facilmente lidos por aplicações como o *Excel* ou *Matlab*, permitindo assim uma fácil construção de estatísticas, para análise dos resultados. O Listagem 11 mostra um exemplo do ficheiro de reporting gerado pela aplicação.

Listagem 11: AppServicesDiscoverProtocol reporting

```
1  SR,Ped1.3,Ped1.1
2  SR,Ped1.3,Ped1.1
3  SA,Ped1.1,Ped1.3,Services: Service1; Service2; Service3.
4  RA,Ped1.1,Ped1.3,Services: Service1; Service2; Service3.
5  SR,Ped1.3,Ped1.0
6  SR,Ped1.3,Ped1.2
7  SR,Ped1.3,Ped1.1
8  SA,Ped1.1,Ped1.3,Services: Service1; Service2; Service3.
9  RA,Ped1.1,Ped1.3,Services: Service1; Service2; Service3.
10 SR,Ped1.3,Ped1.0
11 SA,Ped1.0,Ped1.3,Services: Service1; Service2; Service3.
12 RA,Ped1.0,Ped1.3,Services: Service1; Service2; Service3.
13 SR,Ped1.3,Ped1.2
```

7

TESTES E RESULTADOS

Após finalizada toda a implementação que respeita às comunicações entre atores, é importante realizar testes de forma a perceber o real funcionamento e comportamento do simulador.

Sendo um dos principais objetivos deste simulador a realização de simulações em grande escala, a performance deste é um fator fulcral na sua avaliação, tendo sido por isso efetuados testes de performance ao simulador.

Foram efetuados dois testes para fazer uma análise da verdadeira melhoria de performance efetuada no simulador. Para tal foram realizadas duas simulações, uma à última versão do simulador antes do arranque desta dissertação, e outra simulação já com as devidas alterações implementadas. Dessa forma o primeiro teste efetuado respeita a uma simulação na versão 1.6 do simulador, versão esta de Outubro de 2012, a segunda simulação respeita à versão 2.0 do simulador, versão que contém já as melhorias implementadas. Por fim, a última simulação foca-se no comportamento e análise do simulador face às comunicações implementadas.

7.1 DESEMPENHO BARTUM V1.6 VS. BARTUM V2.0

Tal como já foi descrito no Capítulo 4, foram corrigidos alguns métodos e implementadas melhorias no simulador, com vista a corrigir alguns problemas de performance.

Para avaliar as reais melhorias no simulador, foi elaborada uma primeira simulação à última versão do simulador antes do inicio desta dissertação. Usando o mesmo ambiente de teste, foi elaborada também uma simulação à versão 2.0 do simulador, versão esta já com as alterações para melhoria de desempenho implementadas. Dessa forma torna-se assim possível fazer uma comparação entre estas duas versões, e fazer uma avaliação das

reais melhorias implementadas.

7.1.1 *Ambiente de Teste*

Para realizar os testes a estas duas versões, foram usados seis computadores de laboratório. Estes foram formatados com o sistema operativo Windows 7 64bits, e foi instalada a versão SE 7 Update 40 do Java. A Tabela 11 mostra as principais características destes computadores.

Processador	Intel Core i3 CPU @ 3,20GHz
Memória RAM	4,00 GB RAM
Sistema Operativo	Windows 7 Profissional 64-bit

Tabela 11: Testes: Principais características dos computadores usados

Para além dos computadores, foi também usado um switch HP com 100 Mbps por porta, responsável pela ligação local entre as máquina, estando assim esta rede local isolada de fatores externos. Foram definidos IP's estáticos de 192.168.1.1 a 192.168.1.6.

Foi definida uma máquina para correr o processo do Global Cordinator (192.168.1.1), e as restantes foram atribuídas a Local Coordinators (192.168.1.2 ao 192.168.1.6), havendo assim 5 Local Coordinators na simulação, tal como ilustra a Figura 57.

Para configurar a simulação foi editado o ficheiro de configuração *settings.properties* em cada elemento do simulador. Na configuração da máquina onde correrá o Global Coordinator, terá de estar obrigatoriamente configurada a porta TCP necessária ao TCPServer. Deverão estar também configurados os endereços e portas Multicast, para que estas ligações sejam devidamente estabelecidas.

É ainda no ficheiro de configuração do Global Coordinator que deverão estar definidos os geradores de atores da simulação. Para esta simulação foi optado usar um gerador de atores de cada tipo disponível, pedestres, trams e carros. Para cada gerador é configurado o mapa em que estes se movimentam e a coordenada do mapa onde será feita a geração de atores.

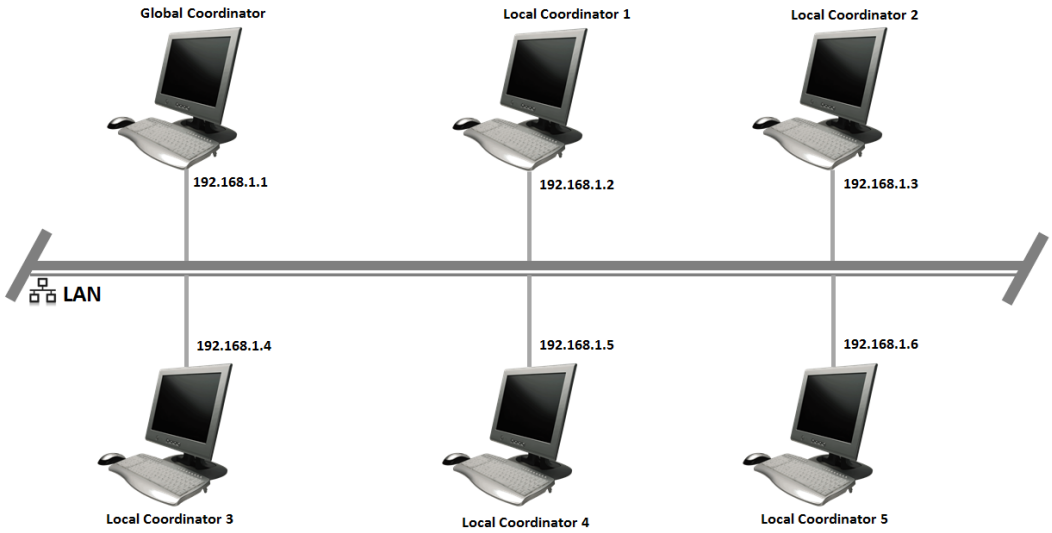


Figura 57: Ambiente de Simulação

Cada gerador foi ainda definido para gerar atores de 1 em 1 segundo, prefazendo assim um total de 3 atores gerados por segundo na simulação.

É ainda necessário definir qual o mapa a usar na simulação. Tendo sido decidido usar um mapa OSM de toda a cidade de braga, representado este uma área urbana já suficientemente grande para a grandeza esperada da simulação. O Listagem 12 mostra as linhas do ficheiro de configuração para a definição do mapa e dos geradores.

Listagem 12: Ficheiro de configuração da simulação

```
#Maps
Map.Number=1
Map.1=Braga-grande.osm

#Genarators
Generator.Number=3

Generator.1=Ped1
Ped1.X=-840102.5363914277
Ped1.Y=4156904.419146793
Ped1.Maps=Map.1

Generator.2=Tra1
Tra2.X=-839986.5188706163
Tra2.Y=4156931.8470627265
```

```
16 Tra2.Maps=Map.1
17
18 Generator.3=Car1
19 Car1.X=-842689.14
20 Car1.Y=-4154968.06
21 Car1.Maps=Map.1
```

No ficheiro de configuração de cada Local Coordinator foi também necessário definir os endereços e portas das ligações TCP e Multicast, para que todas as ligações fossem estabelecidas com sucesso. O Listagem 13 mostra as linhas de configuração destas ligações.

Listagem 13: Ficheiro de configuração da simulação (LCs)

```
1  #GlobalCoordinator IP address and port
2  GlobalCoordinator.IP=192.168.1.1
3  GlobalCoordinator.port=8000
4
5  #Multicast group and port
6  Multicast.IP=228.2.2.2
7  Multicast.port=7070
8
9  #Multicast Message group and port
10 MulticastMessage.IP=228.3.3.3
11 MulticastMessage.port=7171
```

7.1.2 *Simulação BartUM v1.6*

Para a versão 1.6 do simulador, após montado o ambiente de teste foram executados os diferentes elementos da simulação e medidas as suas performances ao longo da simulação. Para tal foi usada a ferramenta "Monitor de Desempenho"(perfmon) do Windows, que permite configurar quais os elementos de medição e a frequência de captação destes, tornando-se assim possível realizar uma monitorização de forma automatizada e por fim exportar os dados capturados ao longo de toda a simulação.

Os testes foram realizados com uma duração suficientemente grande para que se tornasse percetível o comportamento do simulador ao longo do tempo, esta teve assim uma duração de aproximadamente 80 minutos.

Foram então configurados os seguintes elementos para medição:

- Carga de CPU (%)
- Memória RAM em Utilização (%)
- Carga na Rede (bytes/s)

7.1.2.1 *Carga de CPU*

A Figura 58 apresenta um gráfico com a carga de CPU ao longo do tempo no Global Coordinator e vários Local Coordinators presentes na simulação. O eixo das abcissas representa o tempo em minutos e o eixo das ordenadas a carga de CPU em percentagem.

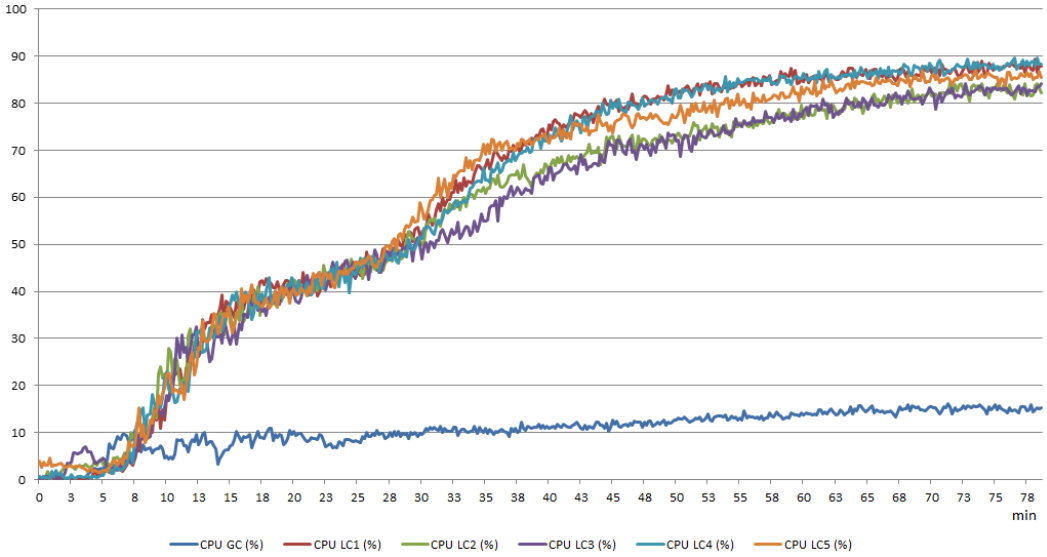


Figura 58: Simulação v1.6 - Carga CPU

7.1.2.2 *Utilização da Memória*

A Figura 59 apresenta um gráfico com a percentagem de memória utilizada ao longo do tempo no Global Coordinator e vários Local Coordinators presentes na simulação. O eixo das abscissas representa o tempo em minutos e o eixo das ordenadas a percentagem da memória da máquina. Este valor não tem grande significado para o simulador, contudo o objetivo é analisar a evolução da memória usada ao longo do tempo.

7.1.2.3 *Carga na Rede*

A Figura 60 apresenta um gráfico com a carga na rede dos vários elementos ao longo do tempo no Global Coordinator e vários Local Coordinators

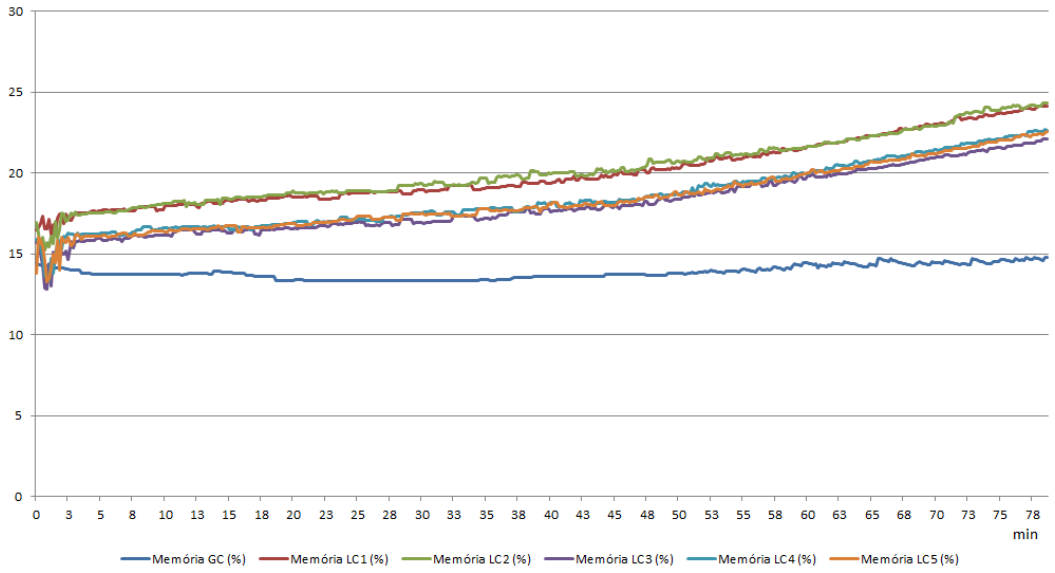


Figura 59: Simulação v1.6 - Utilização da Memória

presentes na simulação. O eixo das abscissas representa o tempo em minutos e o eixo das ordenadas a carga na rede da máquina em (bytes/s), tratando-se do somatório dos dados de entrada e saída de/para a rede por segundo.

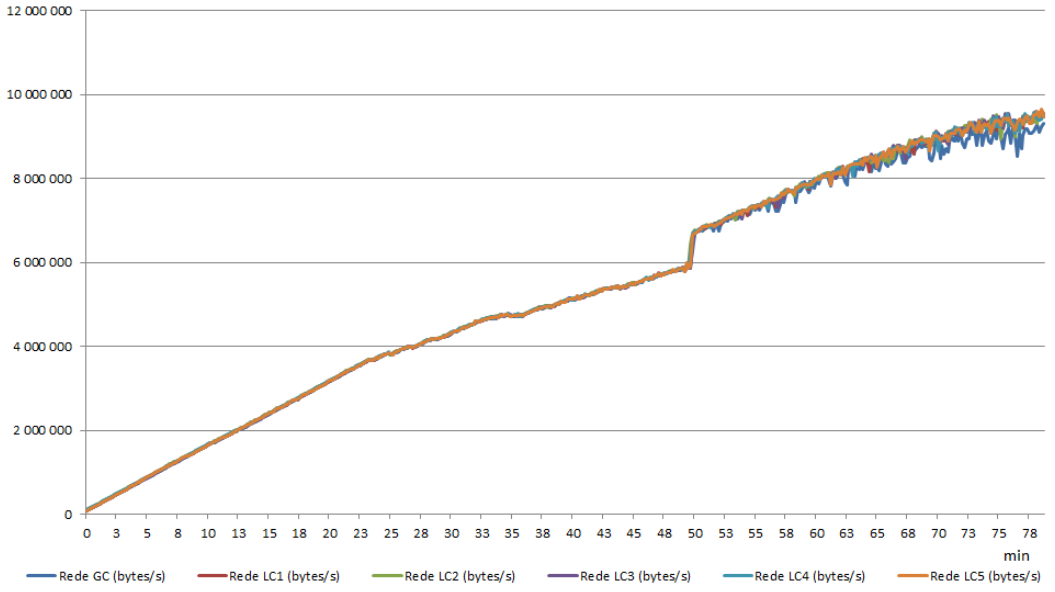


Figura 60: Simulação v1.6 - Carga na Rede

7.1.3 *Simulação BartUM v2.0*

Tal como para a versão 1.6 do simulador, para a versão 2.0 após montado o ambiente de teste foram também executados os diferentes elementos da simulação e medidas as suas performances ao longo desta.

Da mesma forma, os testes foram realizados com uma duração suficiente grande para que se tornasse percetível o comportamento do simulador, desta vez foi percetível mais rapidamente, tendo por isso esta simulação uma duração de aproximadamente 68 minutos.

7.1.3.1 *Carga de CPU*

A Figura 61 apresenta um gráfico com a carga de CPU ao longo do tempo no Global Coordinator e vários Local Coordinators presentes na simulação. O eixo das abscissas representa o tempo em minutos e o eixo das ordenadas a carga de CPU em percentagem.

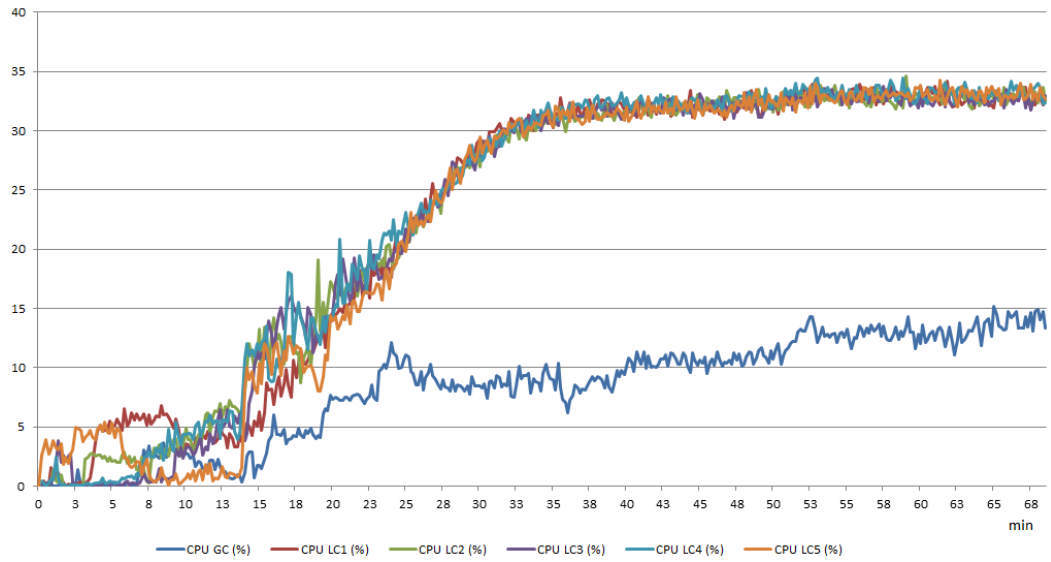


Figura 61: Simulação v2.0 - Carga CPU

7.1.3.2 *Utilização da Memória*

A Figura 62 apresenta um gráfico com a percentagem de memória utilizada ao longo do tempo no Global Coordinator e vários Local Coordinators presentes na simulação. O eixo das abscissas representa o tempo em minutos e o eixo das ordenadas a percentagem da memória da máquina.

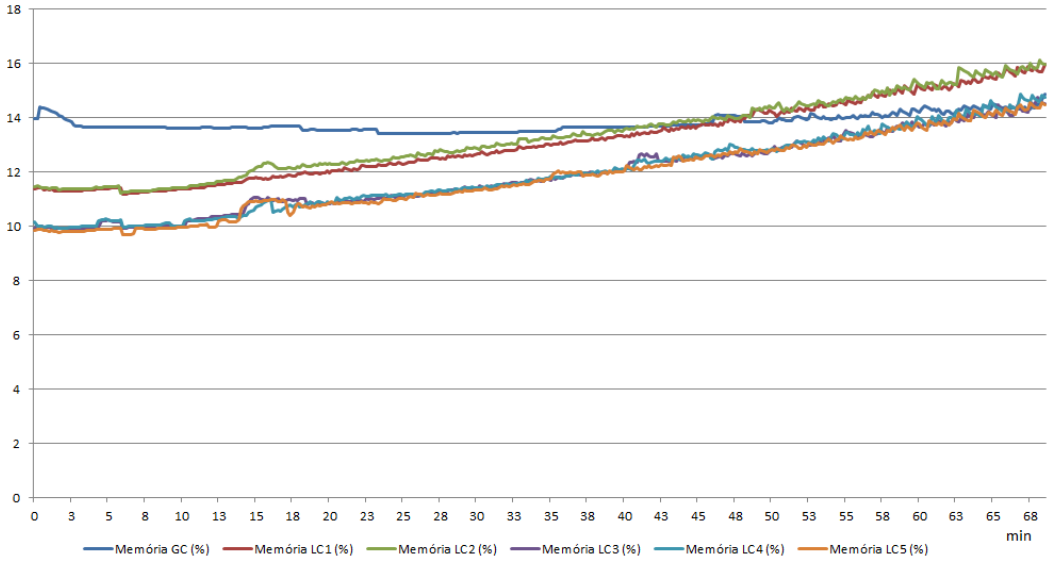


Figura 62: Simulação v2.0 - Utilização da Memória

7.1.3.3 *Carga na Rede*

A Figura 63 apresenta um gráfico com a carga na rede dos vários elementos ao longo do tempo no Global Coordinator e vários Local Coordinators presentes na simulação. O eixo das abscissas representa o tempo em minutos e o eixo das ordenadas a carga na rede da máquina em (bytes/s), tratando-se do somatório dos dados de entrada e saída de/para a rede por segundo.

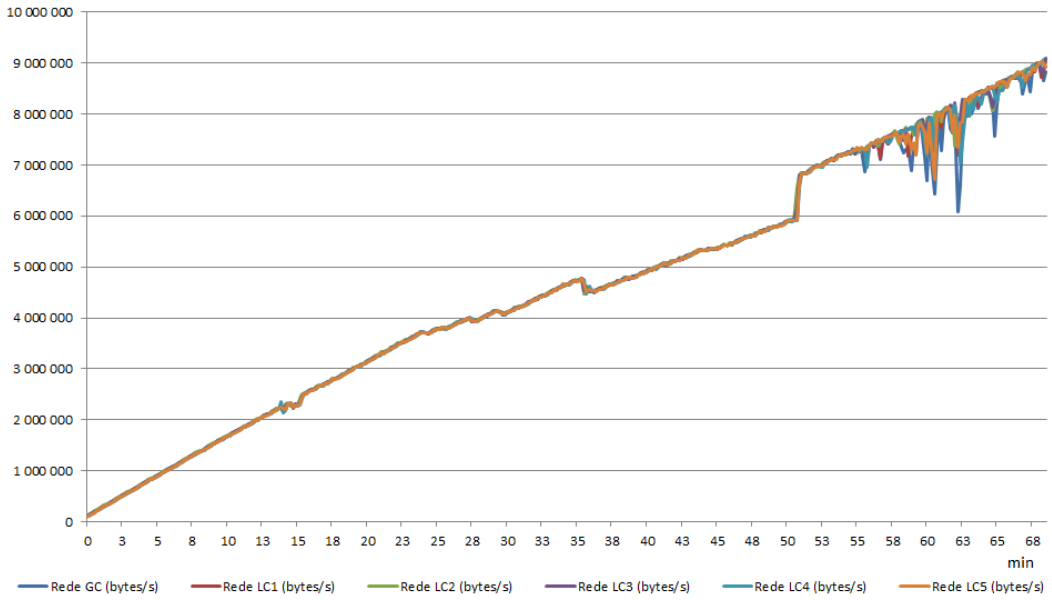


Figura 63: Simulação v2.0 - Carga na Rede

7.1.4 *Conclusões*

Após realizados e comparados os testes às versões 1.6 e 2.0 do simulador, pode-se então tirar conclusões quanto à evolução da performance deste após pensadas e implementadas as melhorias apresentadas no Capítulo 4.

Após a análise da evolução da carga de CPU nas duas versões do simulador 1.6 e 2.0, a primeira perceção que se pode tirar é uma grande redução na carga de CPU dos vários Local Coordinators, logo após a alguma instabilidade inicial na simulação. Também, tal como esperado a carga de CPU no Global Coordinator mantém-se "igual"em ambas versões.

Em análise da evolução da memória em ambas as simulações, pode-se concluir que a memória utilizada tem uma evolução ao longo tempo muito idêntica em ambas as versões, mas registando valores de memória utilizada mais baixos na versão 2.0 em cerca de 65%. Contudo este valor só pode ser tido em conta, sabendo que foram as mesmas máquinas nas mesmas condições a correr a simulação, e que não existiam mais aplicações abertas que pudessem dessa forma estar a influenciar os resultados da simulação.

Em termos de análise da carga de rede ao longo da simulação, pode-se concluir que tal como esperado a carga é idêntica em ambas não existindo dessa forma diferenciações a apontar no comportamento das versões 1.6 e 2.0. Contudo os valores da carga de rede são elevados para os objetivos de escalabilidade do simulador. Sendo que estão a ser gerados nas simulações três atores por segundo, pode-se verificar que aos 75 minutos de simulação, estarão cerca de 13.500 atores presentes na simulação e a rede apresenta já 9 500 000 bytes/s (72.48 Megabits), correspondendo assim aproximadamente a 72% da capacidade máxima da rede local usada. Tal deverá representar um problema na versão 3.0 do simulador, pois já estarão implementadas as comunicações, e aumentará assim o uso da rede.

Pode-se finalmente concluir que foram alcançadas com sucesso as modificações implementadas para a melhoria de performance do simulador. Foi reduzida de forma acentuada tanto a carga de CPU como a memória utilizada.

7.2 DESEMPENHO BARTUM V3.0

Com o objetivo de avaliar a performance do simulador após terem sido implementadas as comunicações e a simulação do protocolo Bluetooth, foi realizada uma simulação para fazer a devida avaliação.

7.2.1 *Ambiente de teste*

Devido à indisponibilidade física no acesso às máquinas usadas nas simulações anteriores, os testes à versão 3.0 foram efetuados em máquinas diferentes, com características diferentes, não sendo assim possível fazer um devida comparação direta com os testes de performance às versões anteriores.

Para este teste foram usadas 4 máquinas com sistemas operativos diferentes mas em todas instalada a versão SE 7 Update 45 do JAVA. A Tabela 12 mostra as principais características destes computadores.

	Processador	Memória RAM	Sistema Operativo
GC	Pentium 2 1.8Ghz	2 Gb	Windows XP 32bits
LC 1	Intel Core i7 2.3Ghz	6 Gb	Windows 8 Pro. 64bits
LC 2	Intel DualCore 1.6Ghz	3 Gb	Windows XP Pro. 64bits
LC 3	Intel DualCore 1.6Ghz	3 Gb	Windows XP Pro. 64bits

Tabela 12: Testes: Principais características dos computadores usados

Para além dos computadores, para os ligar numa rede local foi usado um router SMC WBR14-N4 10/100 Mbps por porta, desligado de qualquer outra fonte para que não existissem perturbações externas na rede. Foram definidos IP's estáticos de 192.168.1.1 a 192.168.1.4.

Foi definida uma máquina para correr o processo do Global Coordinator (192.168.1.1), e as restantes foram atribuídas a Local Coordinators(192.168.1.2 ao 192.168.1.4), havendo assim 3 Local Coordinators na simulação, tal como ilustra a Figura 64.

Para configurar a simulação foram usadas todas as configurações dos testes anteriores, tendo sido definidos três geradores de atores, um de

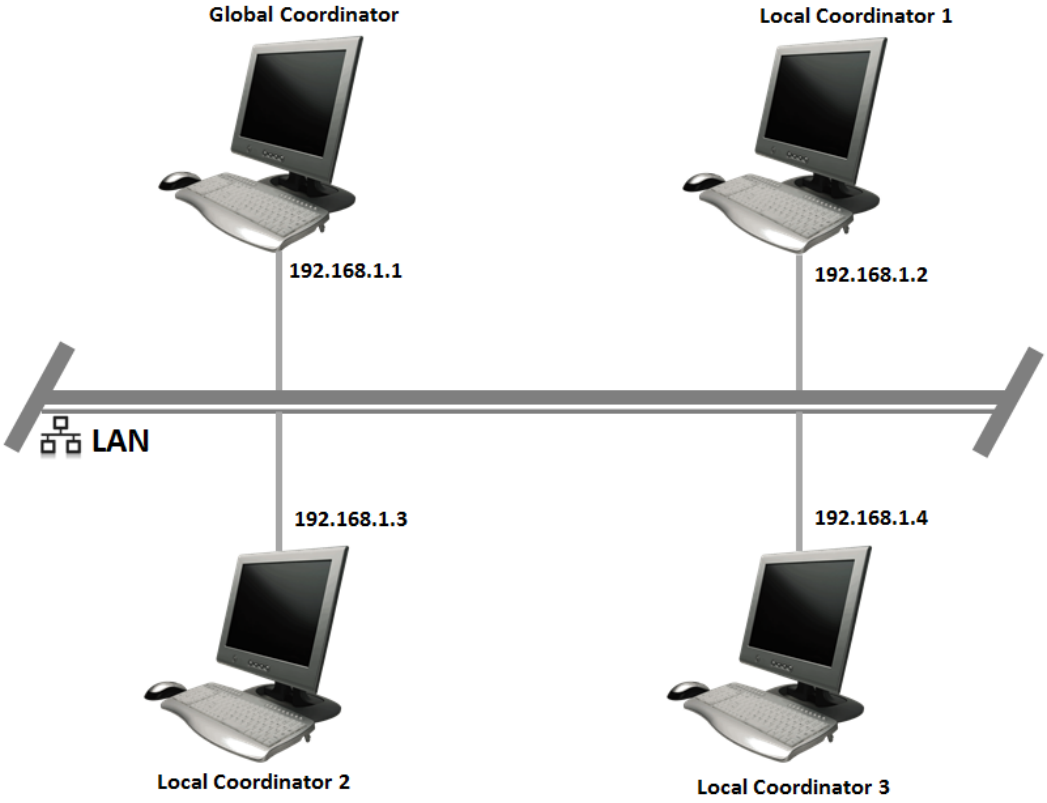


Figura 64: Ambiente de Simulação

cada tipo disponível, pedestres, trams e carros. Foi também definido que os atores gerassem atores de 1 em 1 segundo, prefazendo assim um total de 3 atores gerados por segundo na simulação. Para cada gerador de atores foi ainda associado o mesmo mapa de simulação dos testes anteriores, um mapa OSM de toda a cidade de braga. Para fazer uso das comunicações foi ainda usada a aplicação AppServicesDiscoverProtocol.

7.2.2 *Simulação BartUM v3.0*

Tal como para os testes anteriores, após montado o ambiente de teste foram também executados os diferentes elementos da simulação e medidas as suas performances ao longo desta.

Da mesma forma, os testes foram realizados com uma duração suficiente grande para que se tornasse percetível o comportamento do simulador, tendo sido desta vez percetível aos 55 minutos.

Tal como para os testes anteriores, após montado o ambiente de testes foram executados os diferentes elementos da simulação e medidas as suas performances ao longo da simulação.

7.2.2.1 *Carga de CPU*

A Figura 65 apresenta um gráfico com a carga de CPU ao longo do tempo no Global Coordinator e vários Local Coordinators presentes na simulação. O eixo das abscissas representa o tempo em minutos e o eixo das ordenadas a carga de CPU em percentagem.

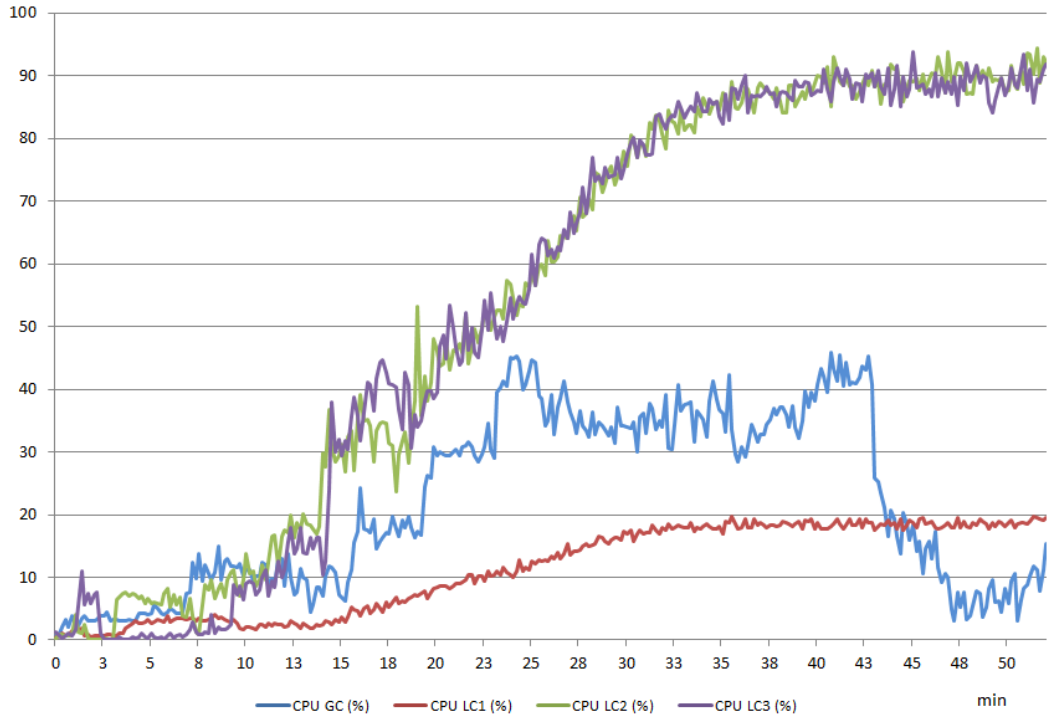


Figura 65: Simulação v3.0 - Carga CPU

7.2.2.2 *Utilização da Memória*

A Figura 66 apresenta um gráfico com a percentagem de memória utilizada ao longo do tempo no Global Coordinator e vários Local Coordinators presentes na simulação. O eixo das abscissas representa o tempo em minutos e o eixo das ordenadas a percentagem da memória em uso da máquina.

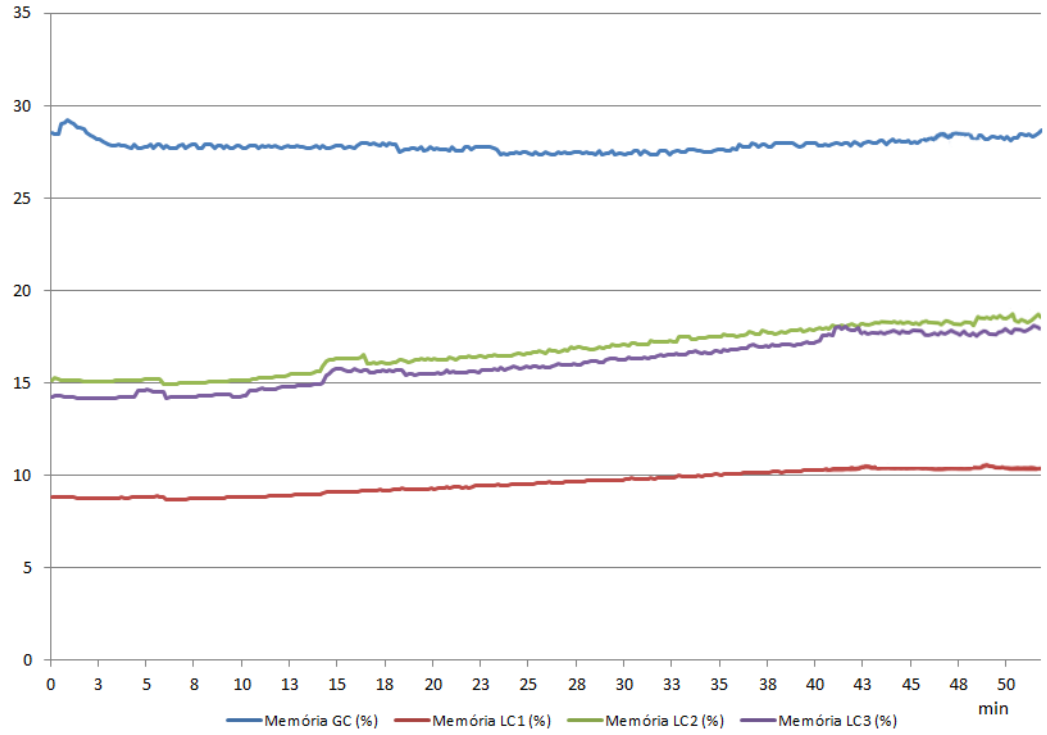


Figura 66: Simulação v3.0 - Utilização da Memória

7.2.2.3 *Carga na Rede*

A Figura 67 apresenta um gráfico com a carga na rede dos vários elementos ao longo do tempo no Global Coordinator e vários Local Coordinators presentes na simulação. O eixo das abscissas representa o tempo em minutos e o eixo das ordenadas a carga na rede da máquina em (bytes/s), tratando-se do somatório dos dados de entrada e saída de/para a rede por segundo.

7.2.3 *Conclusões*

Após realizado o teste ao simulador na versão 3.0, onde já estão implementadas as comunicações e a simulação do protocolo Bluetooth, pode-se então tirar conclusões face à performance do simulador com as comunicações.

A primeira conclusão a tirar é a falha do simulador ao minuto 43 da simulação. Tal facto deveu-se ao excesso de carga na rede e à consequente falha das comunicações entre os vários elementos da simulação. Na versão 2.0 do simulador, este já apresentava indícios de que a rede estaria a ser

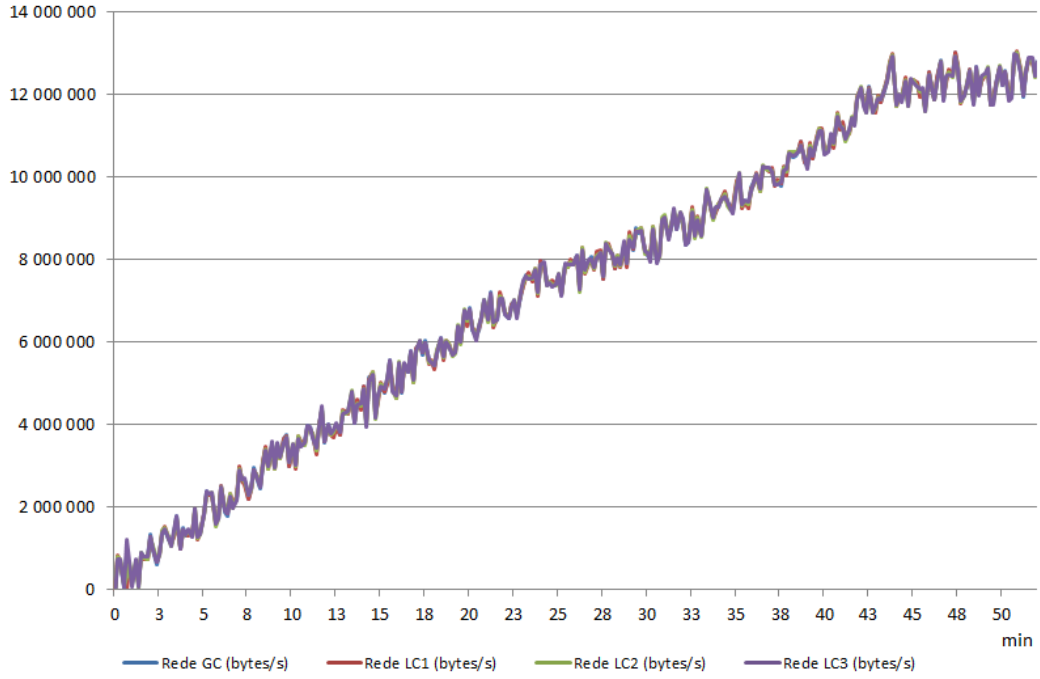


Figura 67: Simulação v3.0 - Carga na Rede

sobrecarregada apenas com a troca de informação das posições dos diversos atores entre os vários elementos do simulador. Após a implementação das comunicações entre atores, implementação do protocolo Bluetooth e consequentes aplicações que geram tráfego, o simulador atingiu assim ao fim de cerca de 7740 atores, a carga máxima da rede local usada.

Com a análise da evolução da carga do CPU durante a simulação, pode-se verificar a baixa de processamento do Global Coordinator ao minuto 43. Pode-se ainda concluir que os Local Coordinators se mostraram robustos face à perda das comunicações, continuando a processar as ações dos seus atores locais atuais.

Visto estarem a ser usadas máquinas com características diferentes, pode-se concluir que o simulador carece de máquinas com bom desempenho de processamento. As máquinas do Local Coordinator 2 e 3, mostraram-se pouco escaláveis, tendo atingido cerca de 90% de processamento aos 43 minutos (2580 atores locais). Contudo o Local Coordinator 1, visto estar a ser executado numa máquina bem mais potente, teve uma carga de CPU bem mais baixa, situando-se apenas em cerca de 18% para o mesmo instante de tempo 43 minutos.

Com a análise da evolução da memória pode-se concluir que esta continua com uma boa performance mantendo valores baixos, e não mostrando um grande crescimento ao longo da simulação. Pode-se também verificar a estagnação do aumento da memória após o minuto 43.

Quanto à rede, tal como já referido, esta atingiu o pico máximo de 100% (13 107 200 bytes / 100 Mbps) aos 43 minutos de simulação. Contudo, manteve-se com um crescimento estável até ao momento de lotação da rede.

Para a resolução deste problema, poderiam ser tomadas algumas medidas com vista a diminuir a carga na rede. Uma das medidas poderia passar pela diminuição da frequência de envio das mensagens de atualização de posições e mensagens das comunicações. O MulticastSender e MulticastMessageSender, estão configurados para fazer o envio das novas atualizações a cada 100 e 1000 milissegundos respetivamente.

Ao aumentar a frequência de envio no MulticastSender, poderão ser criados problemas na movimentação dos atores, visto estes estarem a movimentar-se tendo em conta a posição dos atores vizinhos nesse instante de tempo. Ao aumentar a frequência de envio das atualizações de posição, estes poderiam chocar sem que realmente isso acontecesse na prática, pelo simples motivo da posição ainda não ter sido atualizada. Tal poderia ser resolvido com a criação de um histórico de posições anteriores.

Com o aumento da frequência de envio no MulticastMessageSender, não existem à partida grandes problemas no funcionamento das comunicações, visto ter sido criado um histórico de posições para que estes verifiquem a receptividade das mensagens para o instante de tempo em que estas foram enviadas.

Outra solução possível para a resolução deste problema, passaria pela passagem de noção temporal no simulador, passando de tempo real para tempo de simulação, permitindo assim que não existissem restrições de performance. Contudo esta solução, acabaria com a componente de visualização em tempo real do módulo Vizualization.

8

CONCLUSÕES E TRABALHO FUTURO

Com o enorme crescimento do uso de dispositivos móveis, tornou-se importante que existam aplicações capazes de simular o movimento e a interação entre estes dispositivos. Foi com esta necessidade, e com a falta de aplicações capazes de efetuar simulações em grande escala que surgiu este simulador.

Com o intuito de obter ajuda na implementação do simulador, foi feito um estudo profundo às tecnologias abrangentes e a outros simuladores de mobilidade, obtendo assim conhecimento e ideias para o desenvolvimento deste simulador.

O projeto descrito nesta dissertação de mestrado foi desenvolvido com base no trabalho já realizado no âmbito de outras dissertações de mestrado, tendo sendo por isso um desafio entender todo o trabalho posteriormente realizado, e conseguir trabalhar como evolução deste. Existiram vários problemas de fiabilidade e performance no funcionamento do projeto anteriormente desenvolvido, contudo estas foram corrigidas e ultrapassadas com sucesso.

Foram corrigidos *bugs* do simulador como a leitura não genérica de mapas OSM ou a não possibilidade de escolha de criação de *logs*. Foram também analisadas e implementadas com sucesso, soluções para a correção de problemas de performance que este apresentava, tal como o da função *findNextDestination*.

Como âmbito desta dissertação, foram estudadas e implementadas soluções para a possível comunicação entre atores no decorrer de uma simulação. Foram examinadas possibilidades e testadas com vista a serem implementadas as melhores opções para o simulador, quer por motivos de funcionalidade, quer por motivos de performance.

Foi também implementado um módulo de simulação de uma tecnologia de comunicação, o Bluetooth. Foi estudada a tecnologia, analisada

toda a sua pilha protocolar e pensada a melhor implementação possível para a sua simulação, como módulo deste simulador. Foram ainda criadas aplicações, com vista a usarem as funcionalidades implementadas, e a tornarem assim possível que seja feito um estudo ao real funcionamento dos módulos implementados.

No fim desta dissertação pode-se concluir que o trabalho a que se foi proposto, foi desenvolvido com sucesso, estando a funcionar da forma pretendida. O mecanismo de armazenamento das mensagens em cada Local Coordinator, respondem totalmente às necessidades de funcionamento e performance. Os módulos de distribuição de mensagens entre os vários elementos do simulador, também funcionam da forma pretendida, tratando de forma adequada as mensagens de *broadcast* e *unicast*, sendo fiável e estável durante a simulação. O modulo de simulação da tecnologia de comunicação Bluetooth, foi também implementado com sucesso respondendo aos requisitos de simulação da tecnologia, e não comprometendo o funcionamento e performance do simulador.

No fim deste trabalho foram realizados testes de performance ao simulador, para poder assim obter uma perceção do real funcionamento deste antes e após as alterações implementadas. Os testes correram da forma esperada, mostrando o real funcionamento do simulador após as implementações efetuadas, comprovando os melhoramentos efetuados e o bom comportamento deste, excetuado o agravamento na performance deste quando à carga de rede usada, contudo foram apresentadas possíveis soluções para a resolução deste problema.

Como trabalho futuro a desenvolver neste simulador urbano de movimento e comunicações, seria à partida prioritário melhorar o simulador no que respeita ao desempenho deste com a carga de rede, pois a situação atual compromete o funcionamento deste com simulações em grande escala.

Como continuação deste projeto, outro ponto chave para a evolução do simulador, poderia passar pela implementação de outras tecnologias de comunicação como o Wi-Fi ou NFC.

Será também importante pensar na possível mudança de abstração temporal do simulador, passando este de tempo real para tempo de simula-

ção, resolvendo assim quaisquer problemas de performance deste, mas ao mesmo tempo acabando com a possibilidade de vista de simulação em tempo real com o módulo *Vizualization*.

Outro ponto de evolução possível deste simulador, poderá ser a implementação de módulos estatísticos para avaliar o comportamento das comunicações. Poderiam ser geradas estatísticas do envio e recetividade das mensagens com base nos *logs* gerados ao longo da simulação, ou até em tempo real para serem apresentados no módulo do *Vizualization*.

BIBLIOGRAFIA

[1] Rahul Balani. Energy Consumption Analysis for Bluetooth, WiFi and Cellular Networks. 2007.

[2] Bluetooth SIG. Specification of the Bluetooth System, 2010.

[3] Jennifer Bray and Charles F Sturman. *Bluetooth, Connect Without Cables*. Prentice Hall PTR, 1st edition, 1999.

[4] PALO Wireless Bluetooth Resource Center. Gfsk differences and advantages over fsk modulation, Setembro 2013. URL `http://www.palowireless.com/infotooth/knowbase/radio/109.asp`.

[5] ETSI. Enhanced data rates for global evolution, Setembro 2013. URL `http://www.etsi.org/technologies-clusters/technologies/mobile/edge`.

[6] IEEE Standard for Information technology. IEEE Standard 802.11. Telecommunications and information exchange between systems. Local and metropolitan area networks-specific requirements, 2012.

[7] Hossam Afifi Houda Labiod and Constantino De Santis. *Wi-Fi, Bluetooth, ZigBee and WiMax*. Springer, 1st edition, 2007.

[8] Ari Keranen. Opportunistic network environment simulator. diploma thesis, Helsinki University of Technology, May 2008. 29.

[9] Ari Keranen and Jorg Ott. Increasing reality for dtn protocol simulations.

[10] Peter Vortisch Martin Fellendorf. *Fundamentals of Traffic Simulation*. J. Barcelo, 2010.

[11] Laurent Miranda. Monitorização e registo de movimento em simulação de ambientes urbanos. diploma thesis, Universidade do Minho, October 2012.

[12] Simulation of Urban MObility (SUMO). Sumo - networks/abstract network generation, Julho 2013. URL `http://sumo.sourceforge.net/doc/current/docs/userdoc/Networks/Abstract_Network_Generation.html`.

[13] Fredrik Osterlind. A Sensor Network Simulator for the Contiki OS. 2006.

[14] Rui Pinheiro. Modelos de comportamento para simulação de mobilidade em ambientes urbanos. diploma thesis, Universidade do Minho, October 2012.

[15] Francisco Silva. Simulador de movimento para ambientes urbanos. diploma thesis, Universidade do Minho, October 2010.

[16] Philipp Sommer. Design and analysis of realistic mobility models for wireless mesh networks. diploma thesis, ETH Zurich, September 2007.

[17] Fuqin Xiong. *Digital Modulation Techniques*. ARTECH HOUSE, 2000.